Specification and Verification of Security Policies for Smart Cards

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Herr Dipl.-Inf. Matthias Schwan
geboren am 08.08.1973

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Wolfgang Coy

Gutachter:

1. Prof. Dr. Johannes Köbler
2. Prof. Dr. Ernst-Günter Giessmann
3. PD Dr. habil. Werner Stephan

Tag der mündlichen Prüfung: 13. Februar 2008

Abstract

Security systems that use smart cards are nowadays an important part of our daily life, which becomes increasingly dependent on the reliability of such systems, for example cash cards, electronic health cards or identification documents. Since a security policy states both the main security objectives and the security functions of a certain security system, it is the basis for the reliable system development.

This work focuses on multi-applicative smart card operating systems and addresses new security objectives regarding the applications running on the card. As the quality of the operating system is determined by the underlying security policy, its correctness is of crucial importance. A formalization of it first provides an unambiguous interpretation and second allows for the analysis with mathematical precision. The formal verification of a security policy generally requires the verification of so-called safety properties; but in the proposed security policy we are additionally confronting security properties. At present, safety and security properties of formal system models are verified separately using different formalisms.

In this work we first formalize a security policy in a TLA system specification to analyze safety properties and then separately verify security properties using an inductive model of cryptographic protocols. We provide a framework for combining both models with the help of an observer methodology. Since all specifications and proofs are performed with the tool VSE-II, the verified formal model of the security policy is not just an abstract view on the security system but becomes its high level specification, which shall be refined in further development steps also to be performed with the tool. Hence, the integration of the two approaches within the tool VSE-II leads to a new quality level of security policies and ultimately of the development of security systems.

Keywords:
IT Security, Smart Cards, Security Policy, Formal Verification

Zusammenfassung

Chipkarten sind ein fester Bestandteil unseres täglichen Lebens, das immer stärker von der Zuverlässigkeit derartiger Sicherheitssysteme abhängt, zum Beispiel Bezahlkarten, elektronische Gesundheitskarten oder Ausweisdokumente. Eine Sicherheitspolitik beschreibt die wichtigsten Sicherheitsziele und Sicherheitsfunktionen eines Systems und bildet die Grundlage für dessen zuverlässige Entwicklung.

In der Arbeit konzentrieren wir uns auf multi-applikative Chipkartenbetriebssysteme und betrachten neue zusätzliche Sicherheitsziele, die dem Schutz der Kartenanwendungen dienen. Da die Qualität des Betriebssystems von der umgesetzten Sicherheitspolitik abhängt, ist deren Korrektheit von entscheidender Bedeutung. Mit einer Formalisierung können Zweideutigkeiten in der Interpretation ausgeschlossen und formale Beweistechniken angewendet werden. Bisherige formale Verifikationen von Sicherheitspolitiken beinhalten im allgemeinen den Nachweis von Safety-Eigenschaften. Wir verlangen zusätzlich die Betrachtung von Security-Eigenschaften, wobei aus heutiger Sicht beide Arten von Eigenschaften stets getrennt in unterschiedlichen Formalismen verifiziert werden.

Die Arbeit stellt eine gemeinsame Spezifikations- und Verifikationsmethodik mit Hilfe von Observer-Modellen vor, die sowohl den Nachweis von Safety-Eigenschaften in einem TLA-Modell als auch den Nachweis von Security-Eigenschaften kryptografischer Protokolle in einem induktiven Modell erlaubt. Da wir alle Spezifikationen und Verifikationen im Werkzeug VSE-II durchführen, bietet das formale Modell der Sicherheitspolitik nicht nur einen abstrakten Blick auf das System, sondern dient gleichzeitig als abstrakte Systemspezifikation, die es in weiteren Entwicklungsschritten in VSE-II zu verfeinern gilt. Die vorgestellte Methodik der Integration beider Systemmodelle in VSE-II führt somit zu einer erhöhten und nachweisbaren Qualität von Sicherheitspolitiken und von Sicherheitssystemen.

Schlagwörter:
IT-Sicherheit, Chipkarten, Sicherheitspolitik, Formale Verifikation

Acknowledgments

First of all I would like to thank my advisers Prof. Johannes Köbler from Humboldt-University and Prof. Ernst-Günter Giessmann from T-Systems for their constant and inspiring support during the preparation of this thesis and beyond. To work with them has been very diversified and exciting.

Because this work has been funded by Deutsche Telekom AG and is strongly related to the R&D project *Verisoft*[1] run at T-Systems I had the great chance to experience research in an industrial context. In this regard I would like to express my thanks to Dr. Gunter Laßmann and the whole security group at T-Systems who all gave valuable advice and a fruitful working environment.

My special thanks also go to PD Dr. Werner Stephan from German Research Center for Artificial Intelligence (DFKI) in Saarbrücken for his helpful suggestions and for refereeing this thesis. He and his group, particularly Dr. Georg Rock, Lassaad Cheikhrouhou and Bruno Langenstein, were the main partners in the project and of my work and gave maximum support.

Moreover, without the encouragement of my family and friends this work would not have been finished yet, thank you all.

[1] http://www.verisoft.de

Contents

Contents **ix**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Overview 1
1.2 Overall structure 5
1.3 Publications 6

2 Case studies **7**
2.1 Multilevel access control 7
2.2 Airline loyalty program 9
2.3 Electronic signatures and biometrics 10
2.3.1 Electronic signature requirements 10
2.3.2 Biometric identification systems 11
2.3.3 The BioSig example 12
2.4 Electronic health card 14
2.5 Conclusion 16

3 Smart card security requirements **17**
3.1 Security policies 17
3.2 Security threats for smart cards 19
3.3 Security objectives of smart cards 23

4 Survey of smart card designs **27**
4.1 Java Card 28
4.2 Multos 30
4.3 SMaCOS 31
4.4 BasicCard 33
4.5 Limitations of the security policies 33

5 The extended security policy **37**
5.1 The SMaCOS security model 37
5.1.1 Communication channels 40
5.1.2 Loading new applications 43
5.2 External applications and devices 44
5.2.1 The basic idea 44
5.2.2 Electronic signatures 45
5.2.3 The extended security model 47
5.3 Summary of security functions 49
5.4 Application to the BioSig example 52
5.5 Alternative design decisions 58
5.6 Conclusion 60

6 The analysis of security policies **63**
6.1 Security triangle relations 63
6.2 Application of formal methods 66
6.2.1 Formal models of security policies 67
6.2.2 Formal analysis of cryptographic protocols 70
6.2.3 Safety versus security properties? 72
6.3 Security evaluation criteria 75
6.4 Conclusion 84

7 Safety properties of state-based systems **87**
7.1 The Temporal Logic of Actions (TLA) 87
7.1.1 The basic formalism 88
7.1.2 System verification 93
7.2 A smart card system model 96
7.2.1 System specification 97
7.2.2 Safety properties 102
7.3 Formalization in VSE-II 104
7.3.1 Verification Support Environment VSE-II 104
7.3.2 Formalization of the system components 105
7.3.3 Fairness and concurrency 113
7.3.4 Verification of safety properties 115
7.4 Conclusion 121

8 Security properties of protocols **123**
8.1 Paulson's inductive approach 123
8.1.1 The basic formalism 124
8.1.2 Protocol verification 126
8.2 A smart card authentication protocol 128

8.2.1 Protocol formalization . . . 129
8.2.2 Security properties . . . 132
8.3 Formalization in VSE-II . . . 135
8.3.1 VSE model of the authentication protocol . . . 136
8.3.2 Verification of security properties . . . 140
8.4 Conclusion . . . 143

9 Linking safety and security **145**
9.1 The basic idea . . . 145
9.2 Observer methodology . . . 150
9.3 Extension of the smart card model . . . 153
9.3.1 Cryptographic protocol integration . . . 155
9.3.2 Observer component . . . 159
9.4 New verification tasks in VSE-II . . . 161
9.5 Conclusion . . . 165

10 Summary and conclusion **167**

A State-based specification in VSE **171**
A.1 Model of the SSCM in VSE-SL . . . 171
A.2 Properties of the SSCM in VSE-SL . . . 175

B VSE model of the authentication protocol **177**
B.1 Model of the protocol in VSE-SL . . . 177
B.2 Properties of the protocol in VSE-SL . . . 179

C VSE model of the Extended SSCM **183**
C.1 Defining system states in VSE-SL . . . 183
C.2 VSE-SL Specification of the eSSCM . . . 186

Bibliography **197**

Index **205**

List of Figures

1.1 A multi-applicative Smart Card Operating System 2
1.2 An extended multi-applicative Smart Card Operating System 3
1.3 Formal verification tasks of an extended Smart Card Model . . 4

2.1 Simplified biometric system 11
2.2 Allowed execution chains in the `BioSig` example 13
2.3 Access rights and data flow of the eHealth scenario 15

3.1 Parties involved in smart card based systems 20
3.2 Considered security threats of the smart card system 22

4.1 Overview of the Java Card design 29
4.2 Overview of the Multos card design 31
4.3 Overview of the SMaCOS smart card design 32
4.4 An extended smart card operating system design 35

5.1 Access rights between applications in SMaCOS 39
5.2 Communication channels between applications 42
5.3 Communication channels between applications 43
5.4 Integration of external applications 45
5.5 Electronic signature example 46
5.6 Integration of external signature applications 47
5.7 The extended smart card model 48
5.8 A selection of communication channels between applications in the `BioSig` example 54
5.9 A selection of Communication channels between applications in the `BioSig` example (cont.) 56
5.10 Alternative design of the `BioSig` example using channel programs only 59

6.1 Possible security triangles 65
6.2 Formalization and verification of security policies 69

6.3 Formalization and verification of security properties 71
6.4 Possible formalization and verification tasks 74
6.5 Relationships between TOE representations and requirements 79

7.1 A simple smart card system model 97
7.2 Refined simple smart card system model 100
7.3 VSE Development Graph window of the SSCM 106
7.4 VSE specification window of component `KeyGen` 108
7.5 VSE proof tree window of Lemma 7.3 119

8.1 The mutual authentication protocol 129
8.2 The inductive model of the mutual authentication protocol . . 130
8.3 VSE Development Graph of the mutual authentication protocol 137
8.4 VSE proof tree window of the Confidentiality Theorem 143

9.1 Possible traces of events in protocol verification 147
9.2 A simple smart card system model 148
9.3 Traces of histories in the TLA specification 149
9.4 Verification task on traces 150
9.5 Extended verification task on traces 150
9.6 Relations between Model Specification and VSE-SL Specification 151
9.7 Linking VSE-SL Specifications 151
9.8 Linking VSE-SL Models by means of observer mappings . . . 152
9.9 VSE Development Graph of system states 154
9.10 VSE Development Graph of the extended SSCM 160

List of Tables

2.1 Activation matrix of the `BioSig` example 14

4.1 Comparison of multi-applicative smart card operating systems 34

5.1 Elements of the SMaCOS policy model 38
5.2 Security functions implementing the security objectives and countering the security threats 50
5.3 Assignment of applications to access classes 54
5.4 Assignment of external devices to access classes 56
5.5 Some possible execution chains of the example `BioSig` 58

6.1 Components, families and classes of the EAL's 81

Chapter 1

Introduction

1.1 Overview

Smart cards are present in many application fields of our daily life. One of the most popular examples is the SIM card in mobile phones, which provides an authentication technique in order to activate a cell phone and to authenticate a user in a GSM network. Further examples include pay-TV, electronic toll[1] or electronic cash systems[2]. At present, credit cards based on magnetic stripes are being replaced by the integrated EMV chip card technology. Smart cards also serve as secure signature creation devices in compliance with the European Signature Directive. Moreover, there is a broad variety of ongoing large-scale projects that will have an impact on our daily life like electronic health cards, job cards, electronic travel documents and ID cards.

In all applications above mentioned, the smart card itself serves as a secure storage device managed by the calling application that is running on a host computer. The smart card gains access to an internal secret, for instance a secret cryptographic key, once the user is authenticated by a personal identification number (PIN). This limitation to a passive component is due to processing power and storage capacity. Future generations of smart cards will take advantage of higher processing power, which will allow them to play a more active role in security applications. There is an increasing need of more sophisticated operating systems of smart cards that can manage several applications on the card and the communication between each other. This describes the transition from the *multi-functional smart card* to the *multi-applicative smart card*.

A few multi-applicative smart card operating systems have been devel-

[1] http://www.toll-collect.de
[2] http://www.geldkarte.de

oped that implement different security policies, with regard to the confidentiality and the integrity of applications running on the card as well as the loading of new applications onto the card, for example Java Card[3], Multos[4], Windows for Smart Cards (no longer supported) or Basic Card[5]. The reliable communication between an on-card application and the outside world has to be accomplished by the application itself, which makes the on-card applications always accessible from the outside world. Security critical information may leak by revealing what applications are stored in the card. Moreover, even the absence of a particular application on a card may provide an attacker with helpful information about the version of the underlying smart card operating system that might be vulnerable to specific attacks. The accessibility is particularly critical for contactless smart cards (RFID) as it is hard for the smart card holder to detect the access attempt.

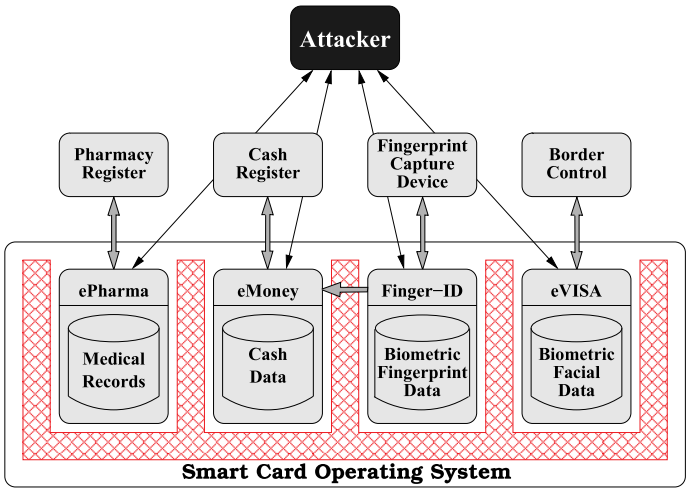


Figure 1.1: A multi-applicative Smart Card Operating System

Figure 1.1 illustrates the framework of a general multi-applicative smart card operating system providing firewall mechanisms for the separation of on-card applications. In the example the application `ePharma` implementing an electronic receipt system and the electronic visa application `eVisa` do not allow any communication with other applications on the card. The electronic cash application `eMoney` may require the biometric authentication of the user before the payment. The external communication is managed by each application itself.

[3]see http://java.sun.com/products/javacard
[4]see http://www.multos.com
[5]see http://www.basiccard.com

In fact, security matters arise if an on-card application needs to communicate with external devices. For instance, a biometric identification application that stores the reference template on the card and provides an on-card matching unit needs to get the actual biometric user data from a capture device like a fingerprint sensor. The overall security of the application does not only depend on the quality of the matching algorithm but on external devices. One of the best-known attacks is to fool a fingerprint identification system with artificial fingers. For this reason, there is an increasing need of powerful aliveness checks of the biometric capture device together with a secure transmission of the actual biometric user data from the sensor to the matching unit. In the end, the overall security level of the biometric identification application highly depends on the external devices.

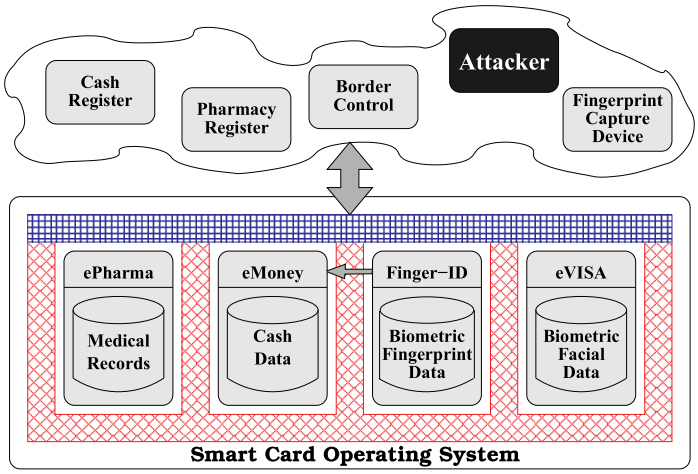


Figure 1.2: An extended multi-applicative Smart Card Operating System

Such security requirements are also desired for smart card terminals (smart card readers). Following our example, we consider the biometric identification application in the context of an electronic passport that is equipped with a contactless smart card holding the biometric application as well as other applications. We only want the border control to be able to access and even to *see* the biometric application. Nobody else shall identify the smart card as an electronic passport while scanning for RFID cards. In other words, it is the particular smart card terminal that *activates* or *hides* particular applications on the card. We want the operating system to act as a firewall between applications on the card as well as between on-card applications and the outside world. The extension of the firewall is shown in Figure 1.2 (blue).

The shift of the managing of external devices or external applications from the applications themselves into the operating system may not only lead to

an increased security level but also significantly reduce development costs of the applications, as the applications can take advantage of the extended operating system functions. This is of special interest if the application providers strive for an evaluation under the Common Criteria (CC) or ITSEC scheme: the less complex is an application, the less is the effort to achieve a successful evaluation. Therefore, the application providers depend on the accuracy of the security functions provided by the operating system. The formal specification and formal analysis of the functionalities can help increasing the reliability of the operating system.

Security engineering regards an operating system as the implementation of a security policy stating the security objectives and security functions of the system, which therefore requires the careful analysis of the security policy. Here, we work out an extended security policy for multi-applicative smart cards that combines internal and external security functions, as shown in red and blue (respectively) in Figure 1.2. Since confidentiality and integrity between on-card applications can be achieved by access control mechanisms fully controlled by the operating system, the mechanisms are not applicable between on-card applications and external applications. Those communications over an open network cannot be fully controlled by the operating system. Other mechanisms like cryptographic protocols have to be considered. In Figures 1.2 and 1.3, the red colored functions denote access control mechanisms and the blue colored functions denote cryptographic mechanisms.

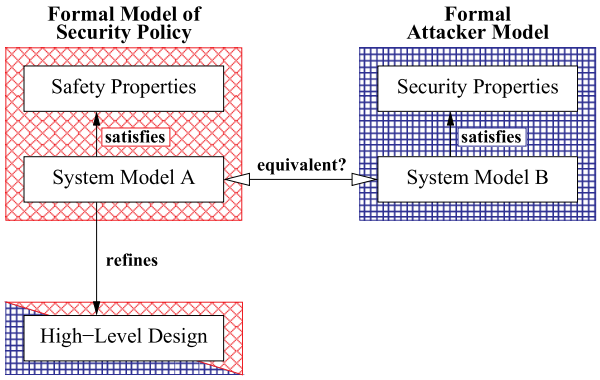


Figure 1.3: Formal verification tasks of an extended Smart Card Model

The formal analysis of the different security mechanisms is generally treated separately as shown in Figure 1.3. Formal models of security policies mostly formalize the access control mechanisms in a transition system and verify that the system never enters insecure states, which is commonly known as proving safety properties. The formal analysis of cryptographic protocols may also prove that no insecure states are reachable but additionally take

an attacker with well defined skills into account. To stress the difference we refer to the latter as proving security properties.

In the overall development process of the smart card operating system the transition system is generally refined in a high-level system design that inherits only the safety properties. Therefore, we have to formally demonstrate that the high-level design also inherits the security properties, either by showing that it additionally refines the system under the attacker model or by showing a kind of equivalence between the two system models. In this work, we choose the second approach and provide a practical framework for formal verification of safety and security properties considering the extended security policy for multi-applicative smart cards. The framework involves a powerful tool support by VSE-II (Verification Support Environment) that allows for a top-down approach of the system development starting with the security policy as the abstract system design and following with subsequent refinement steps.

1.2 Overall structure

First, an introduction of four significant applications is presented in Chapter 2. These applications involve a physical access control, a purchaser's loyalty point system, an electronic signature creation with biometrics and electronic health cards. One may regard them as an ideal of what smart cards should be able to perform. Based on the case studies, we develop security requirements for smart card operating systems that are described in Chapter 3. Then, we briefly analyze existing operating system designs (Java Card, Multos, BasicCard and SMaCOS) and the underlying security policy in order to investigate their suitability for the proposed case studies in Chapter 4.

In Chapter 5, we introduce the extended security policy and show that the security functions implement the security objectives and counterfight the identified security threats. Our extended security policy takes advantage of the proposed SMaCOS security policy and integrates external devices and external applications. We additionally define an execute-access right that has been regarded as not security relevant to date. Furthermore, we apply the extended policy in one case study, which specifically shows that integrity access levels are indeed very helpful for designing real world applications. The relevance of such integrity access levels has been underscored so far.

Chapter 6 deals with the analysis of security policies in terms of security triangle relations. We investigate how formal methods can help to strengthen the analysis, which leads to a missing link between the formal treatment of security policies and the formal analysis of cryptographic protocols. A

methodology bridging this gap is the topic of the next chapters.

In Chapter 7, we first develop a generic model of a simple smart card operating system that concentrates on the external communication functionality in order to formalize the model in temporal logic. We verify the formal model by proving invariant properties of infinite behaviors of the system model, and show that the verified system model insufficiently covers the intended functionality of external communications.

Cryptographic protocols provide adequate functionalities for the secure communication between several participants over open networks. Chapter 8 introduces a smart card protocol that we formalize and verify by applying the inductive approach by Paulson. The last challenge is the combination of both formal models, which we perform with the help of an observer model methodology as described in Chapter 9. We prove that every behavior of the simple smart card model represents a valid protocol trace.

All formalizations and verification tasks are performed in the interactive theorem prover VSE-II. With this methodology we make sure that a further refinement of the temporal logic model in a high-level design of the system fulfills both safety properties of the system model and security properties of the protocol model. The refinement step itself is not performed in this work. All formal specifications and properties are given in Appendices A, B and C.

1.3 Publications

The extended model of security policy proposed in Chapter 5 has been published in the proceedings of the *ACM Symposium on Information, Computer and Communications Security* (Sch07). An instantiation of the generic simple smart card model has been used in the development and `ITSEC E4 high` evaluation of the T-TeleSec Key Generator (GS06). A cryptographic protocol in the context of a biometric smart card that was partially used in Chapter 8 has been discussed at the anniversary of working group *Sicherheit – Schutz und Zuverlässigkeit* of Gesellschaft für Informatik (LS06). The formal verification of the protocol with the help of VSE-II was published in the proceedings of the *Conference on Computer Safety, Reliability and Security (SafeCOMP2006)* (CRS+06), and parts of the formal proofs are also described in Chapter 8.

Chapter 2

Case studies

The following sections describe four applications in which smart cards play an active role. A multilevel access control system is first introduced followed by an example of an airline loyalty program that has been published in (KAT00b). Next, we will have a close look at the smart combination of an electronic signature application and biometrics. The last example deals with an electronic health system. The aim of this chapter is to present a variety of existing and potential applications that take advantage of enhanced multi-applicative smart cards. Because we are going to use the signature example in later chapters again, it is explained in more detail than the other examples.

2.1 Multilevel access control

The first example considers a simple multilevel access control application `MultAC` that, for instance, gains physical access to a specific room only if the particular user had already accessed the building within a specific time. In general, it must always hold: before access to something is gained all required preconditions have to be fulfilled. This is called *multilevel access control*. Example of useful preconditions are:

C1: access after a hardware is presented (e.g. a smart card);

C2: access after biometric authentication;

C3: access after a pass phrase is presented;

C4: access after the person successfully accessed another access point;

C5: access after a specific room has been left;

C6: access if a specific time limit is not exceeded;

C7: access in compliance with the "four eyes principle".

The list above mentioned is not exhaustive but shows the range of possible preconditions. An access control system may use one or a combination of them.

As an example, we assume a company that defined several security zones in their company grounds, which are the *public zone*, the *low*, *medium* and *high restricted* zone. The public zone is open to everybody and serves as an exhibition and advertisement room. The low restricted zone may be the park garage, the lift lobby and offices. The medium restricted zone hosts the development departments, whereas the data center is classified as the high restricted zone. Every employee gets a smart card labeled with the company logo; it functions as a company and authentication card. Additionally, the card may provide payment functionality for the staff restaurant.

Every employee has to present its valid smart card (`C1`) in order to enter the low restricted zone. The employees of the development department get access to the medium restricted zone after presenting the valid smart card (`C1`) and a biometric authentication (`C2`). Furthermore, they need to have successfully entered the low restricted zone (`C4`). When they leave their department, they have to present the smart card again (`C1`). The successful check-out must take place before the employee can enter the medium restricted zone again (`C5`). We assume the same condition in order to access the data center. Besides, the security policy of the company states that no engineer is allowed to enter the data center alone. Thereby, two smart cards must be presented at the same time (`C7`) in order to get access to a data center. Afterwards, every engineer has to be biometrically verified.

So far, we only addressed a physical access control. The access to computer resources should also be managed by the smart card. Hence, the login routine of the desktop computers uses the smart card and a pass-phrase (`C3`) to authenticate the user. Additionally, a user gets access to the computer only if he has already entered the low restricted zone.

The described scenario may sound very strict or even paranoid and one may say that this scenario is highly unreal. In fact, most of the attacks of company values are done from inside due to poor security mechanisms. Besides, similar systems are commonly implemented having a central control and therefore all access points need to be online or connected. That makes such systems quite expensive, especially in large-scale company grounds. Hence, the main challenge compared to existing systems is to shift the intelligence from a centralized system to the smart card.

The described scenario can be realized with a multi-applicative smart card that holds different applications according to the described access preconditions. The applications must communicate with external terminals and other applications on the card. For instance, application `ACmed` implementing the access control for the medium restricted zone of the development department has first to check if the particular employee has entered the low restricted zone. This could be done via a read access of application `ACmed` to the on-card application `AClow` implementing the access control to the low restricted zone. Application `ACmed` could read a credential from `AClow` and invalidate it after use. Then `ACmed` should be able to execute a biometric application `Bio` on the card implementing a biometric user authentication, which itself needs to communicate with an external biometric sensor. Finally, application `ACmed` has to authenticate itself to the physical door in order to get the door open and hence to grant access to the development department. In this way all described requirements of the `MultAC` application could be implemented.

2.2 Airline loyalty program

In the second example `AirPts`, we assume an airline `A` running a loyalty program and partnering with two hotel chains `H1` and `H2` and two car rentals `R1` and `R2` as proposed in (KAT00b). The airline customers get a smart card and will earn loyalty points when they fly with the airline, stay in one of the partner hotels or rent a car from one of the car rentals. The hotel chains and car rentals run their own loyalty program but would prefer to share a common smart card for their different loyalty programs. Because both car rentals as well as both hotel chains are competitors, they should not see the loyalty points of each other. Besides, the hotel should get the possibility to give extra points if a customer flew with the airline. Furthermore, every partner is allowed to give own loyalty points and to collect customer's information for own marketing purposes. Such information is confidential and is not shared with any of the other companies.

Based on this requirements, the software that manages hotel `H1` loyalty points must behave differently from the software that manages hotel `H2` loyalty points. This is because the hotels have different policies, whereby the customer earns extra points whether he flew with the partner airline or rented a car before. In addition, it must be possible to update the software in order to allow limited time special offers, e.g. stay five times in one month and earn 500 bonus points. Finally, the introduction of new partners should be feasible, for instance a credit card company that provides credit card functionality

and gives loyalty points if a flight is paid with the credit card.

To implement the `AirPts` example we create one on-card application for every partner `A`, `H1`, `H2`, `R1` and `R2`. In this scenario applications `H1` and `H2` shall be able to read points from application `A` but not mutually. Furthermore, the multi-applicative smart card must provide mechanisms for downloading new temporary applications. The overall application could be implemented in the same fashion as the access control example.

2.3 Electronic signatures and biometrics

The `BioSig` example combines the creation of electronic signatures with the biometric authentication of the corresponding user. We first give an overview of electronic signature requirements and then introduce biometric systems in order to explain the `BioSig` example.

2.3.1 Electronic signature requirements

The European Parliament and the Council define in Directive 1999/93/EC (DIR99) a framework for electronic signatures in order to facilitate their use and to contribute to their legal recognition. The directive distinguishes between the *advanced electronic signature* and the *qualified electronic signature*, whereby each of them has to fulfill different requirements. An advanced electronic signature (`AeSig`) is an electronic signature that meets the following requirements: (a) it is uniquely linked to the signatory; (b) it is capable of identifying the signatory; (c) it is created using means that the signatory can maintain under its sole control, and (d) it is linked to the data to which it relates in such a manner that any subsequent change of the data is detectable.

A qualified electronic signature (`QeSig`) is an advanced electronic signature that is based on a *qualified certificate* and is created by a *secure signature creation device* (`SSCD`). The main differences between an `AeSig` and a `QeSig` application is the use of a `SSCD`, which holds the signature creation data, for instance a cryptographic key. The `SSCD` creates a signature only after the successful authentication of the signatory. This can be done by a knowledge based method (PIN or Password) or by biometric methods (SSGS00). The successful evaluation of the `SSCD` according to the Common Criteria is compulsory (PP01). The `SSCD` together with the user authentication method shall provide a *strength of function (SOF)* of `high`. A detailed introduction to the Common Criteria is given in Section 6.3.

Additionally, the combination of a knowledge based method of strength `SOFhigh` and a biometric method of strength `SOFmedium` is in regard to the

directive an opportunity, which means once the signature creation data are *activated* with the `high` method the signature can be created after the successful authentication of the signatory with the `medium`-method. The number of such signature creations may be limited. So far, there are no technical realizations available that combine both authentication methods.

In the following chapters we name all other non-evaluated devices (according to the Directive) *signature creation devices* `SCD`. In other words, the `SSCD` provides evaluated (pre-checked) security functionality whereas the `SCD` provides claimed security functionality. Hence, a qualified signature is legally seen as more reliable than an advanced electronic signature.

2.3.2 Biometric identification systems

The purpose of a biometric system is the authentication of users by means of behavior or physiological characteristics of the individual user. The major components of a biometric system are the capture device for capturing the biometric data (a fingerprint sensor), an extraction unit for extracting the individual characteristic from the raw data called biometric template (the minutia of a fingerprint), the matching unit for comparing the actual biometric template with the already stored biometric reference data, which is a further component. The biometric reference data have to be created initially in order to perform a biometric authentication. This initial procedure is called *enrollment*.

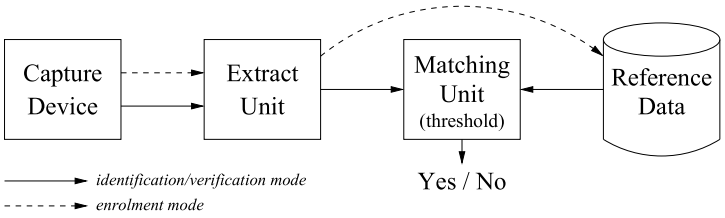


Figure 2.1: Simplified biometric system

The procedure of a biometric authentication distinguishes two modes: in the biometric *verification mode* the biometric template is compared to a single reference data, whereas in the biometric *identification mode* the biometric template is compared to a set of reference data. The biometric system outputs a positive authentication if in verification mode the matching score of the one-to-one comparison is higher than a preset threshold. In identification mode the highest matching score of the one-to-many comparisons with all reference data must be higher than the preset matching

score for a positive authentication of a user. The components and the different modes of operation of a simplified biometric system are illustrated in Figure 2.1. A detailed introduction to biometric systems can be found in (MMJP03; NTN02; Zha00; Las06).

The performance of biometric systems is generally measured in terms of *false rejection rate (FRR)* and *false acceptance rate (FAR)*, which state the rate of false rejected users and false accepted impostors. It is a unique characteristic of biometric systems that these rates will never be zero. From the security point of view the FAR is of special interest. The *Biometric Evaluation Methodology* of the Common Criteria (CC02) claims a false acceptance rate of:

FAR $= 0,000001$ (1 in 1.000.000) for `SOFhigh`,

FAR $= 0,0001$ (1 in 10.000) for `SOFmedium` and

FAR $= 0,01$ (1 in 100) for `SOFbasic`.

These definitions make sense in a verification mode of a biometric system. For identification mode the SOF will depend on the size of the database. Currently, there is one Protection Profile (PP) of the Common Criteria regarding biometric verification mechanisms (BSI04a) published that claims an *EAL2* with `SOFbasic`.

But the overall performance of a biometric system cannot be simply expressed in terms of error rates. Other vulnerabilities have to be taken into account (Las02). The majority of attacks to fool a biometric system is to present faked characteristic to the capture device as it has been demonstrated with artificial fingers (MMYH02; TKZ02). That is why powerful aliveness checks of the biometric capture device are needed as well as a secure transmission of the actual biometric data from the capture device to the matching unit (WSE04). We want the smart card to perform the biometric matching and to confidentially store the biometric reference data as well as to control the confidential transmission of the captured biometric data.

2.3.3 The BioSig example

In our example `BioSig` a smart card is considered that serves as a `SCD` as well as a `SSCD`. One may think of a PGP key that a user likes to store in the same smart card as his secret signature key that corresponds to a qualified certificate. In this scenario the former application belongs to a `AeSig` and the latter one belongs to a `QeSig`. Furthermore, an `EAL4high` evaluated external signature application `ExtSig` running on a host computer

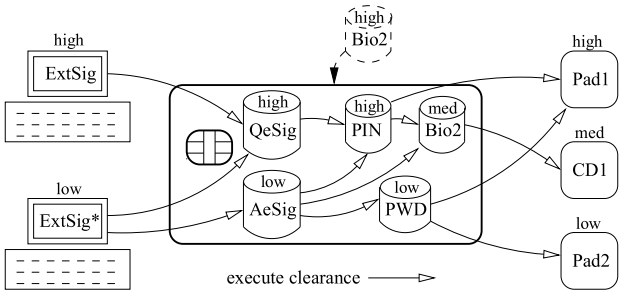


Figure 2.2: Allowed execution chains in the `BioSig` example

may execute the signature creation application `QeSig` running on the smart card. Our aim is to prevent high evaluated external signature applications to use the not evaluated `AeSig` application. In addition, we assume a second external signature application `ExtSig*` without any evaluation, e.g. a PGP signature tool.

We already mentioned the different requirements of the signature creation applications `QeSig` and `AeSig` concerning authentication methods. That is why we distinguish a password authentication method `PWD` that is not evaluated at all and an authentication with a personal identification number `PIN` that is `EAL4high` evaluated. Please note, that the `QeSig` key can not be activated with the `PWD` method. Furthermore, there are two more applications that implement biometric systems `Bio1` and `Bio2`, whereas `Bio1` is also `EAL4high` evaluated while `Bio2` is `EAL4medium` evaluated.

Application `PWD` activates `AeSig` whereas applications `PIN` and `Bio1` activate `QeSig`. Additionally, we say after the successful authentication of the signatory with `PIN` or `Bio1` the signature creation is allowed after the successful authentication via `Bio2` at a maximum of 10 times in succession. We also want to give the opportunity to activate `AeSig` with the higher evaluated authentication applications.

All authentication applications need user interaction and thus depend on external devices, e.g. PIN pads or biometric capture devices. Thereby, the biometric applications `Bio1` and `Bio2` on the smart card need to check whether the biometric sensor (capture device) meets the individual requirements of the biometric system or not. A similar requirement must be postulated for the `PIN` application because it is important that the application communicates with a trusted PIN pad. At present, smart card terminals with integrated PIN pads are developed, which ensures that the PIN can not be altered or eavesdropped while transmitted. In the `BioSig` example we distinguish a trusted PIN pad `Pad1` and an untrusted PIN pad `Pad2`, e.g. the keyboard of a PC. In Figure 2.2 the intended execution chains in our example

	PIN (EAL4h)	PWD (–)	Bio1 (EAL4h)	Bio2 (EAL4m)	PIN or Bio1 with Bio2
AeSig	x	x	x	x	x
QeSig	x		x		x

Table 2.1: Activation matrix of the BioSig example

scenario are given. The Figure also shows that new applications, for instance Bio1, shall be loadable onto the card even the card has already been issued. After the successful installation of Bio1 it gets the same access rights as the PIN application. Hence, the QeSig application can either execute the PIN or Bio1 application for user authentication. Table 2.1 shows the dependencies of the different authentication applications activating a signature creation application. In conclusion we say that the smart card must be able to authenticate its environment in order to decide if specific security requirements are met by the external devices. Additionally, we do not restrict the use of the authentication applications to the BioSig application. They should also be available for other potential applications like access control systems mentioned in the MultAC example.

2.4 Electronic health card

In this section we consider an electronic health card that is issued by a governmental institution to every people. The card grants access to a central database that holds all medical records of the patient. The main goal of the system eHealth is to provide medical staff with the medical history of a patient. This may reduce costs, because some medical checks do not need to be done twice. It is very import to keep this data private. To do so we assume three kinds of cards: a *patient card*, a *physician card* and a *pharmacist card*.

The patient card contains four applications whereas the first application AC controls the access to the medical records in the central database. The second application Pharm can be either used by the physician or by the pharmacist. It serves as a container for prescriptions. The third application Data simply holds public data that are freely accessible, e.g. the name or the blood group of the patient. Furthermore, the patient takes advantage of a loyalty program of an health care company. She earns points when she buys medicine, remedies or other products of this company. The application implementing the loyalty program LP can be loaded onto the card even it is out in circulation. One may think of more temporarily installed applications on

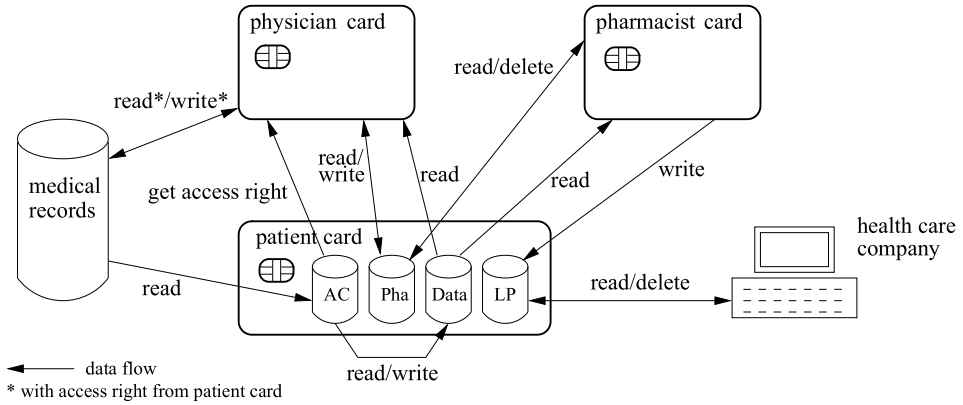


Figure 2.3: Access rights and data flow of the eHealth scenario

the patient card, like a hospital phone card application for in-patients.

We will next define the access rights to the various data. A summary of the overall access rights of all participants is illustrated in Figure 2.3. We assume application `AC` holds authentication data that are unique for every specific card and thus for every patient, like a secret cryptographic key. The authentication data can be activated after the successful authentication of the patient (the card holder), whereby the patient can alternatively choose a pass-phrase based method or a biometric verification. The patient can get a *read only* access to his medical records if he presents his card and authenticates himself to the card. In addition, the patient can set a second pass-phrase or can biometrically enroll a second person in order to define a representative who acts on behalf of the patient. After a successful authentication, the patient also gets *read only* access to the prescription records and *read* and *write* access to the general data. Only the patient has write access to the general data and determines which data are public.

The medical records of a patient shall only be accessible to physicians that have previously been approved by the patient. Therefore, the physician first needs to get the patient's permit, which requires the interaction of the physician card with the patient card. The physician has to activate his physician card via a pass-phrase based method or via biometric verification. Upon successful activation of the physician card and the approval by the patient card, the physician can issue prescriptions for the patient and store them on the patient card. He can also read prescriptions that are already stored on the card, for example those issued by another physician.

For buying medicine the patient presents his card with the stored prescriptions to the pharmacist. The patient card grants a *read* or *delete* access only to the pharmacist card. Therefore, it has to authenticate those cards.

The pharmacist activates the pharmacist card via a pass-phrase method or via biometric verification. In addition, the pharmacist and the patient take part in a loyalty program of a healthcare company. The pharmacist can store loyalty points on the card, e.g. if the patient buys products of this particular healthcare company. The patient can cash the loyalty points in one of the branches of the healthcare company.

As in the other examples, the `eHealth` case study also shows a high level of interconnections between on-card applications and external applications that may run on other smart cards.

2.5 Conclusion

In this chapter, we present four case studies from different application fields that take advantage of a multi-applicative smart card. In all scenarios the smart card plays the central role and shall be able to hold various applications on the card and to manage the communication between the on-card applications as well as the communication to external applications and devices. We emphasized the importance of the integration of external devices in a smart card design as a security element with an example taken from biometric user authentication and electronic signature creation.

Another example could be the development of electronic passports providing a biometric authentication application. The latest generation of German passports provides the photograph of the holder in an electronic way via an integrated RFID chip. Protection is achieved by a cryptographic protocol and the encrypted transmission of the biometric data. In our scenario we want to assure that only the smart card terminals of the authorities are even able to *see* the passport application. As long as the smart card operating system has not authenticated the right terminal, it will not give any information about the application (even that the application exists on the card).

The case studies motivate a new approach of access control of smart cards. As long as confidentiality or integrity between applications is concerned, from an on-card application point of view there shall be no difference whether an application is hosted on the same smart card or outside the card.

Chapter 3

Security requirements for multi-applicative smart cards

When integrating smart cards in security applications as given in the previous chapter, the smart card shall fulfill various security requirements defined by the particular security application. The identification and categorization of common security requirements for smart cards is the aim of this chapter. Typically, the requirements analysis results in establishing a security triangle built up of identified security threats faced by security objectives, which are implemented by security functions countering the security threats. The security objectives together with the security functions are commonly known as the security policy of a system.

In this chapter we first give an introduction to security policies that is further discussed regarding the security triangle and analysis techniques in Chapter 6. Next, the common security threats and resulting security objectives for multi-applicative smart cards are identified. Furthermore, we give an overview of possible security functions that implement the intended security objectives. In the next chapter we will investigate existing smart card security policies whether they fit our defined security policy or not.

3.1 Security policies

The term *security* very often causes confusion when used by different people with different background. Generally, a *security system* provides *security functions* that somehow shall protect specific assets. A *security policy (SP)* clarifies from what the assets are protected and the more interesting question: Which *security threats (ST)* are not covered by the security system? A good general introduction to the terminology can be found in (Eck04; And01). We

define:

> The *security policy* of a system or an organizational unit defines a set of technical and organizational rules, guidelines and responsibilities for the particular entities in order to achieve the defined *security objectives*.

The security objectives are a part of the security policy and state the protection properties of a system. They are generally the result of a *threat analysis*. For multi-applicative smart cards we give the threat analysis and the identified threats in the next section. In other words, a security policy defines "secure" for a particular system taking the security objectives and the security functions (the rules) into account. Therefore, what is secure under one policy may not be secure under a different policy. The intended security objectives can be achieved by using organizational or technical measures. As an example we consider a security objective demanding that only authorized employees are allowed to run a particular application. A typical technical rule, or *security enforcing function*, is the authentication of every user before granting access. Authentication could be realized via a password based method. We call the realizations of security functions *security mechanisms*, which are generally not in the scope of the security policy. The corresponding organizational rule shall state who is allowed to run the application according to the business work flow.

A security policy can be a combination of informal and highly mathematical statements. In order to analyze the correctness of a security policy we use structured representations or abstract models of the security policy that focus on the technical rules. We define:

> A *security policy model (or security model)* is a semi-formal or formal representation of a particular security policy in order to analyze correctness properties.

If we consider a computer system to be a finite state machine with a set of transition functions that change states, then a security policy model is a statement that partitions the states of the system into a set of *secure* states and *nonsecure* states. In this manner a secure system is a system that starts in a secure state and cannot enter a nonsecure state. The more formal introduction to security policies can be found in (Bis03; Gol99).

In general, we distinguish *access control* security models and *information flow control* security models, whereby access control models found widely acceptance in practice. They are additionally divided into *discretionary policies*, *role-based policies* and *mandatory policies*.

In an discretionary access control system the individual user has to set the access rights to its objects in order to allow or deny access of other users or processes. In role-based systems the access rights to objects are predefined and assigned to particular roles. A role defines an individual task and hence, has the clearance to all objects and resources that are needed to solve the task. So, a user is not directly cleared for accessing objects but assigned to a particular role. Lastly, mandatory access control systems predefine all access rights of users to objects that cannot be altered by the individual user. In such systems it is very common to assign all users and objects to different security classes, e.g. `unclassified, confidential, secret`. The access rules are set according to the security classes. We will go into more detail in Section 5.1. Those systems are also called *Multi Level Security System (MLS)*. Generally, in security engineering we first have to identify the security threats that should be countered by a particular security policy that we do for multi-applicative smart cards in the following section.

3.2 Security threats for multi-applicative smart cards

In history smart cards have been introduced to serve as secure embedded system devices in order to store sensitive data and to run sensitive applications. To achieve security we are concerned with the *hardware*, the *operating system* and the *applications* of the smart card. All three components must be designed in a way that they altogether provide the intended security level of the overall system. The operating system functionality depends on the security mechanisms provided by the hardware. Furthermore, the applications rely on the security mechanisms of the operating system. We miss the aim if we on one hand heavily secure the application level or the execution of operating system commands, and on the other hand it is easy for an attacker to read out single storage registers at the hardware level. Therefore, we distinguish attacks on a *physical* and on a *logical* level. In the following, we only consider the logic level and assume the hardware to be secure. In general, there are many parties that have to put into account when analyzing and categorizing attack scenarios. In Figure 3.1 the involved parties are given we are concerned with.

The *cardholder* is the party who uses the smart card and its applications. The cardholder may control some data on the card, but usually has less control of protocols, applications and hardware configurations as it is in contrast known from a personal computer. The system administration is

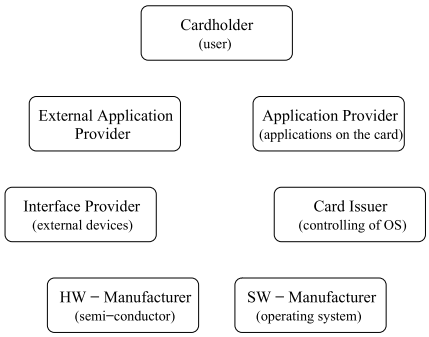


Figure 3.1: Parties involved in smart card based systems

rather controlled by the *card issuer* using the operating system running on the card. This party is also responsible for initial data stored on the card, like authentication keys or signature verification keys. The card issuer differs from the *application providers* that run their applications on the card. In some cases that might be the same company, which means such company acts in two different roles.

In addition, an application provider may act as an *external application provider*, but we again distinguish two different roles. As an example one may think of an ATM of bank A and of bank B where the two different ATM's can be seen as external applications. The smart card serves as an identification card holding an ATM identification application approved by both banks. In the scenario the external application also includes the smart card *interface*, which is the card slot in the ATM. But we can identify many more external devices that serve as interfaces. One may think of card slots in doors for physical access control or biometric capture devices or even special displays that trustfully show the current money balance stored on the card.

Furthermore, we distinguish the *hardware manufacturer* (card manufacturer) and the *software manufacturer* (operating system manufacturer). This distinction makes sense from the security point of view, because all parties might be a potential threat to each other. That is why attacks are also categorized in terms of point in lifetime of a smart card. We can identify attacks in the *design phase* (hardware design of the chip or software design of the OS), in the *production phase* (production of the semiconductor wafer) and attacks while *using* the smart card (running an application). A very good and comprehensive classification of possible attacks can be found in (RE02; SS99).

Since we are concerned with the execution of several applications on a multi-applicative smart card in parallel, we assume a secure hardware as well as a secure development environment. In other words, the operating

system shall prevent attacks in the *using phase*. In contrast, we do not trust the applications (and therefore the application providers) and see the applications as potential threats for other applications whether they are running on the card or outside the card as external applications. In addition, we do not trust the interface devices. They may alter, delete or even create new data while transmitting them to and from the smart card.

We already gave a few application examples in Chapter 2. All examples have in common that many parties are involved in the overall applications. In contrast to a single-application card, where in general the card issuer also designs the applications, in the proposed scenarios the card issuer may differ from the application provider. In the first example `MultAC` the company acts as the card issuer and therefore may perform administration tasks. The individual access control system may be provided by different security companies. These companies may differ from the payment provider that is in cooperation with the staff restaurant. In the `AirPts` example the airline acts as the card issuer and the hotel chains and car rental companies provide their individual loyalty point applications. In the example the airline acts as an application provider as well. We find similar participants in the other examples, whereby each mentioned example application defines its own security requirements and requires a multi-applicative smart card communicating with other applications.

A typical *threat model* considers a malicious application that somehow gains access to data of an application that are supposed to be confidential. Those malicious applications can be considered on the card as well as outside the card. Furthermore, no application should be able to unauthorized alter and delete other applications or even to load new (malicious) applications. We follow the threat analysis in (SRSS99) and summarize all considered security threats in the *threat classification tree* given in Figure 3.2.

`ST1`: *Loading illegal applications* – applications that are not approved by the card issuer or the cardholder may be loaded onto the card and may interfere with already loaded applications, which is not intended, e.g. Trojan horses.

`ST2`: *Pretense of wrong identity by the interface* – an external device may pretend an identity that claims to be a trustworthy device, but in fact the interface device may eavesdrop or manipulate data (see also `ST3`).

`ST3`: *Transferring illegal data* - Data may be altered, eavesdropped, deleted or even created while transmitting it to and from the smart card.

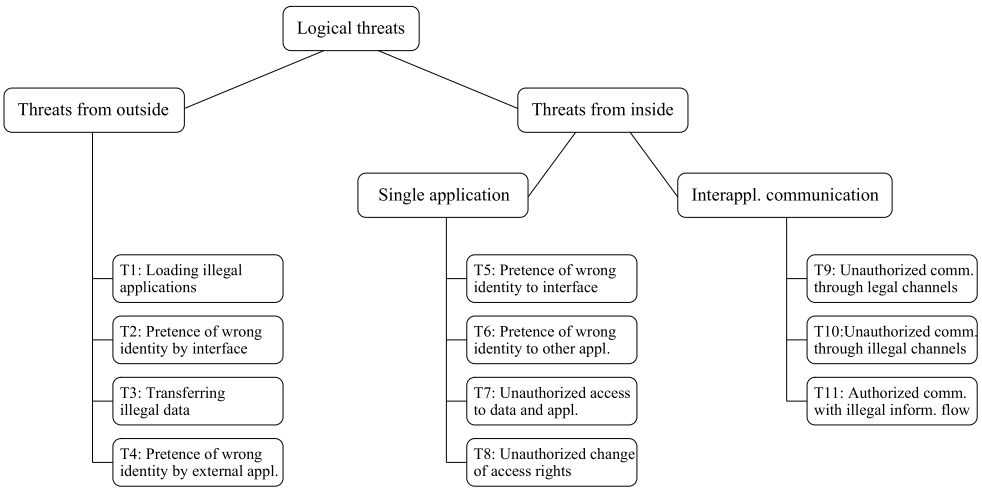


Figure 3.2: Considered security threats of the smart card system

ST4: *Pretense of wrong identity by external applications* – an external application may pretend an identity that claims to be a trustworthy application, but in fact the external application may eavesdrop or manipulate data (see also ST3).

ST5: *Pretense of wrong identity to the interface* – an application running on the card may pretend an identity to an interface device that claims to be trustworthy (in opposition to ST2).

ST6: *Pretense of wrong identity to other applications* - an application running on the card may pretend an identity to applications or external applications that claims to be trustworthy (see also ST5).

ST7: *Unauthorized access to data and applications* – An application may access data or execute applications that it is not authorized to.

ST8: *Unauthorized change of access rights* – An application may make data or applications accessible to other applications that are not authorized to access the data.

ST9: *Unauthorized communication through legal channels* – There may be communication between applications through channels provided by the operating system even if one application provider of the applications does not agree with it.

ST10: *Unauthorized communication through illegal channels* – there may be illegal channels through which there might be unwanted communication

flow between applications.

ST11: *Authorized communication with illegal information flow* – there might be an authorized channel between applications, but the information transmitted is not the information intended to be transmitted via the channel whether in direct or indirect way.

As already mentioned, the identified security threats shall be countered by a particular security policy of a multi-applicative smart card that is discussed in the next section.

3.3 Security objectives of multi-applicative smart cards

In the previous section we identified various security threats that should be countered by the security policy of the smart card. Therefore, we formulate *security objectives* that should be implemented by the security functions of the smart card operating system. The security objectives are a part of the overall *security policy*. We informally give the security policy for multi-applicative smart cards countering the security threats identified in the previous section.

> ***Security policy:***
> *The operating system of the multi-applicative smart card provides functionalities to run several applications in parallel on the card that have been approved by the card issuer to load onto the card. If not agreed by a specific application no other application can access the data or execute the specific application. Communication between applications on the card and outside the card must be agreed by the applications and approved by the card issuer. If demanded by an application only approved external devices are accepted by the operating system.*

We can summarize the following security objectives:

SO1: Approved applications only, e.g. by the card issuer, must be loadable onto the card even the card is in circulation.
(*download of appl.*)

SO2: The applications stored on the card should neither be observable nor alterable by other applications.
(*appl. isolation*)

SO3: The access rights of applications can not be altered by other applications as well as by the applications themselves.
(*appl. control*)

SO4: Applications on the card can communicate, e.g. share files, with each other only if they got the appropriate right. This includes the execution of other applications.
(*comm. within the card*)

SO5: Applications on the card can communicate with applications outside the card (external applications) only if both got the appropriate right, e.g. applications on other cards. This includes the confidentiality and integrity of the transmitted data if requested by one of the applications.
(*external appl. comm.*)

SO6: An application on the card can communicate with external devices only if both got the appropriate right. This includes the confidentiality and integrity of the transmitted data if requested by the application.
(*external device comm.*)

In general there are three ways in operating multi-applicative smart cards as considered in (Gir99). One way is to keep the card and its integrity under the full control of the card issuer. All application providers have to negotiate and agree with the issuer's security policy and implementation guidelines. The card issuer may evaluate applications before loading them on a particular smart card.

The second way is user oriented and assumes a user that buys a blank card from a manufacturer of his choice and plays the issuers role. Hence, the whole card is under the user's control and the user buys applications from providers. One may think of open source projects.

In a third scheme a new participant is added that acts as a trustworthy third party, a certification authority. Those authorities may evaluate the issuer and its smart card and postulate the compliance with a specific security policy. Based on this policy an application provider as well as the user are able to decide whether the issuers policy meet their requirements or not. The application itself may also be evaluated by the third party in order to postulate the compliance with the issuer's guidelines. This scheme is a refinement of the first scheme. We will discuss this scheme in the context of the Common Criteria in more detail in Section 5.4.

In the following we will generally discuss possible *security functions* that implement the security objectives. We can identify two security layers, which are the operating system security layer and application security layer. We

again assume the hardware to be secure. The former one is responsible for the quality of operating system services, like the separation of the individual applications and basic input/output services. The latter layer relies on the operating system security and has to trust it. The applications assume the operating system security to act as it is supposed to. On the other hand, it has to protect all applications even if one application acts in an aggressive way. Therefore, the security objectives SO1 and SO2 are in the responsibility of the operating system. The security objective SO3 states that the access rights of every application is under the full control of the operating system.

Unfortunately, if we do not allow any communication or file sharing between applications at all the overall functionality of the card will be very limited. In order to control access rights to data two main types have been introduced: *discretionary* and *mandatory* access control. The first type is based on the individual application and is the most widely known. The owner (or subject) of an object, e.g. a file, constrains who can access it by allowing particular other subjects to access. A mandatory access control is enforced by the operating system. Neither the target subject nor the owner of an object can determine whether access is granted. We say the former access control is *application driven* while the latter one is *operating system driven*. One can also think of combining both.

If we consider a discretionary access control one should be aware that nothing can prevent application B, which grants access to object a from application A, from copying the information into another object shared with application C, even if C is not allowed to access object a. Hence, information is leaked from A to C. According to our examples the security objective SO4 should be implemented by a mandatory access control.

The security objective SO5 and SO6 can be interpreted as an extension of objective SO4 by taking the communication between on-card and external applications into account. If demanded by either an on-card application or an external application the data must be full of integrity or transmitted confidentially. In addition an on-card application must be able to authenticate an application outside the card and vice versa.

The identified security objectives must be implemented by proper security functions that we discuss in more detail in Section 5.

Chapter 4

Survey of smart card operating system designs

We already mentioned that currently smart cards are widely used as secure storage devices with a variety of functionalities. We call those cards multi-functional smart cards and define multi-applicative smart cards in the following. Most of the native smart card operating systems (in accordance to ISO 7816) do not support the execution of new functions that are loaded after the card has been issued in spite of the fact that it is technically possible. The way of doing this is to load *native code* onto the card. Because most of the smart card micro-controller do not provide a hardware memory access management, every loaded functionality has basically access to every memory unit. This leads to the need of a very careful evaluation of the function before loading it onto the card. This technique is mostly used to fix bugs within the operating system. We do not consider such scenarios. Nevertheless, even if an evaluated application provider is allowed to load a new functionality in its *dedicated file* the problem remains. The most promising solution is the integration of a hardware memory management unit as it has been done for the Infineon SLE88 together with a formal model in (OWL03).

This leads to two main differences between a multi-functional and multi-applicative smart card.

- A multi-applicative smart card allows the loading of new applications (which might be small functions) even the card has been issued.
- Applications running on a multi-applicative smart card may execute or communicate with other applications on the card.

The difference is not determined by the execution of a piece of data, a function or an application on the card by an external application but by the smart card operating system providing the two mentioned services.

A second way of loading and executing applications on a smart card is the use of *interpreters*. An interpreter controls the access to the memory units while executing an application. One of the main advantages of the approach is the more platform independency from the application point of view. On the other hand significantly more processing power and memory capacity is needed to run both the interpreter and the application. We briefly introduce the main operating systems and the implemented security policies in this field from a very abstract point of view. A very good overview of smart card operating systems is given in (RE02).

In this chapter we first have a look at the security policies of Java Card, Multos, SMaCOS and Basic Card in order to motivate an extended security policy given at the end of the chapter.

4.1 Java Card

Java Card technology enables smart cards to run small applications, called *applets*, that employ Java technology[1]. Since we are interested in the security aspect, we briefly introduce the architecture of the Java Card Platform, which is graphically displayed in Figure 4.1. On the lowest level we have the *operating system* that may implement native methods like communication protocols, cryptographic routines and a file management. Please note, a file system according to ISO 7816 is not implemented in this architecture. Such file system can be simulated in an applet, which is necessary if one aims at staying conform with traditional host applications.

The *Java Card Virtual Machine (JVM)* interprets the machine independent byte-code, which is the code obtained after Java compilation. It is mainly responsible for Java-language security and thus, it checks and interprets every line of the byte-code, e.g. for type and syntax correctness, and generates the individual machine code. Due to limited processing power and storage capacity the check may partly be done by an off-card virtual machine called *Java Card Converter*. The converter generates a *Card APplication file (CAP-file)* that can be loaded onto the card.

The *Application Programming Interface (API)* is the standardized interface to card specific features that may be implemented in native code. Together with the JVM it is part of the *Java Card Runtime Environment (JCRE)* that mainly controls the isolation of the individual applets. The Java Card standard does not cover management issues of the card and the applets, like loading and deleting of applets. These issues are addressed by

[1] see: http://java.sun.com/products/javacard/overview.html

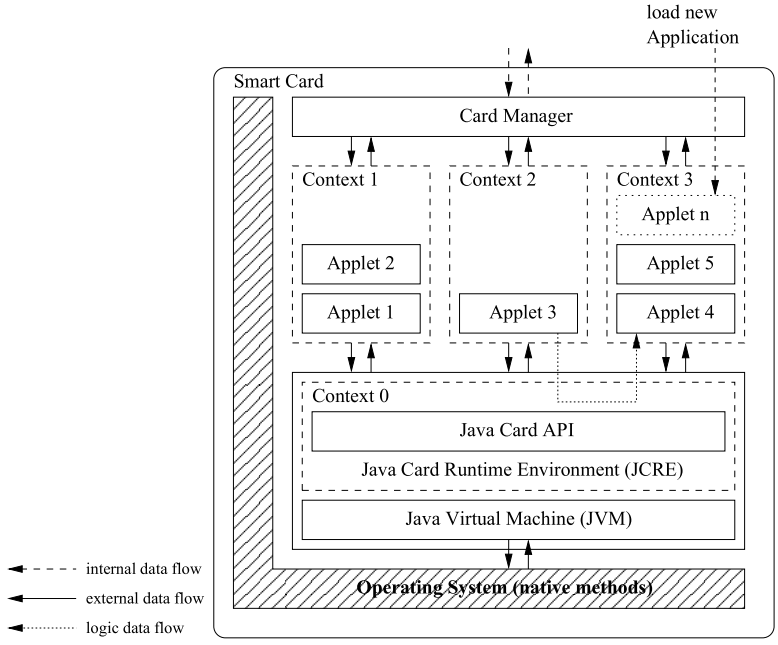


Figure 4.1: Overview of the Java Card design

the Global Platform[2] specification that defines a life-cycle of the card and an additional component called *Card Manager*.

The Card Manager controls the life-cycle of the card and is responsible for loading new applications and for managing the inputs and outputs. It receives an APDU from the smart card terminal and either selects and starts an applet to let the command be executed or forwards the command to the already running applet. Once the execution is finished the Card Manager forwards the responds APDU to the card terminal. The execution of security critical operations, like loading of new applets, requires the authentication at the Card Manager with the pre-set symmetric Card Manager Keys.

The Java security model provides two main features, which are the *Java-language security* and the *applet isolation principle*. The former feature concentrates on safety properties of the language like type safety. Contrary to, for instance C++, Java does not allow *casts* that allow a programmer to write an expression telling the compiler to treat an integer as a pointer. This is a powerful tool for implementing device drivers but may lead to break the type safety. Because we are interested in the main security principles from a more abstract point of view we are not going into more detail here. Additionally, the integrity and authenticity of new applets can be assured via an electronic signature created by the converter and verified by the JVM. The

[2]see: http://www.globalplatform.org/

signature signals the type correctness of the certain CAP-file.

The latter feature is also known as the *sandbox model* and is central for multi-applicative smart cards. It prevents the objects that were created by one applet from being used by another applet. Therfore, every applet is assigned to a particular space called *context*. Applets within one context have mutual access to all objects in the context. Two contexts are strictly divided, which is controlled by the JCRE. The JCRE itself is a special context that can be accessed by all applets of any context. The inter-application communication between two applets of different contexts is possible via *shared interfaces*. If an applet wants to make a method available to other applets of other contexts it explicitly has to give the permit for the method. Once the access right is granted it is not reversible and the method can be invoked from any context. A formal verification of the applet isolation property is given in (ACL03). The authors verify the confidentiality property in an extended JCRE using the Coq proof assistant.

The isolation property is limited to applications installed on the card. In (HV00) a mechanism is developed that allows an application running an a card to dynamically grant access rights to another application either stored on the same card or outside the card. The *JCCap* is based on software capabilities implemented in tickets and controlled by the application. A similar approach is given in the *Gateway model* of the Java Electronic Commerce Framework (Gol97).

4.2 Multos

The *MULTi-application Operating System (Multos)* is controlled by the consortium company MAOSCO Ltd. that promotes and develops the Multos specifications as an open industry standard[3]. Multos technically follows the same basic idea as Java Card technology but is focused on secure execution of code as well as secure loading of new applications onto the card. Therefore, it uses a sophisticated key management within a PKI because Multos is designed to satisfy a special business model in electronic payments. Furthermore it is `E6high` evaluated under the ITSEC scheme, which particularly requires a formal analysis of the underlying security policy.

Multos defines both the *Multi-Application Operating System (MAOS)* and the virtual machine called *Application Abstract Machine (AAM)*. The MAOS provides basic functionality like transmission protocols, cryptographic algorithms and a file manager. The machine independent program code *Multos*

[3]see: http://www.multos.com

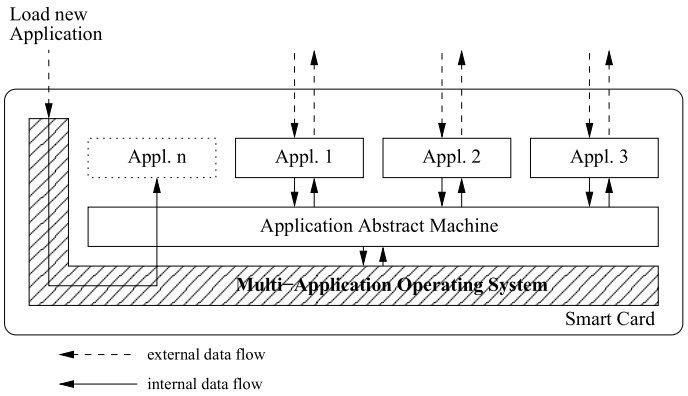


Figure 4.2: Overview of the Multos card design

Executable Language (MEL) is interpreted by the AAM. Usually the programs are developed in C and compiled into MEL. All applications and application data are strictly separated (Figure 4.2). A policy that provides inter-application communication does not exist. Additionally, Multos is a single-threaded operating system that allows only one application to be executed at any time. Since all specifications of the operating system are confidential no further information regarding the security model can be given.

4.3 SMaCOS

The SMaCOS project aims at developing a secure multi-application smart card operating system that provides application isolation as well as the controlled sharing of data. Additionally, the secure download of new applications onto the card is supported. The project is run by IBM and Philips semiconductors. The security model focuses on a very sophisticated policy of access rights to objects (KAT00a; KAT00b). It combines the Bell/LaPadula model for data confidentiality and the Biba model for data integrity. A formal model of the security policy and its verification is given in (SRS+00; SRSS99; SRS+02). In spite of the fact that the SMaCOS takes advantage of the hardware support for supervisor/user mode of the Phillips SmartXA chip the security model is hardware independent on the abstract level. That is why we will have a closer look at the security functions. The general design is shown in Figure 4.3.

In (SRSS99) the security objectives of the security policy are given that are relevant to counter some of the security threats mentioned in Section 3.2 on Page 19.

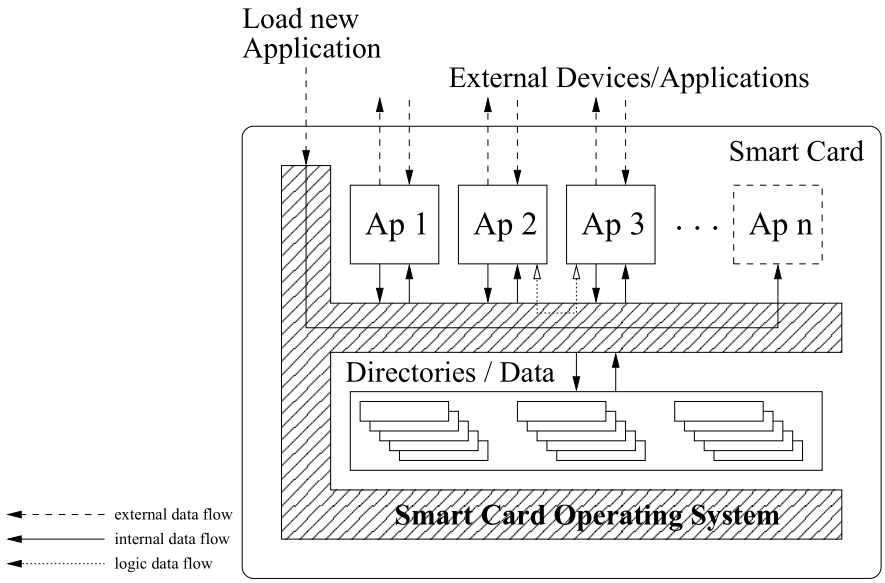


Figure 4.3: Overview of the SMaCOS smart card design

1. *Agreement of the card issuer:* Every application that is loaded onto the card must have the card issuers consent.
2. *Agreement of applications to communication:* Each application that is potentially influenced by a new application must agree on it.
3. *Identity of applications:* Applications on the card must carry an unchangeable information about their identity and access rights when executed.
4. *Authorized access only:* Access to data files is restricted by an appropriate access control scheme.
5. *Authorized changes of access rights only:* Changes to access rights are subject to a mandatory security policy.
6. *Controlled communication:* Communication between applications takes place only if they agree with it.
7. *Secure communication:* Communication between applications must be neither observable nor subject to manipulations by other applications.

The security objectives are implemented by mandatory access control mechanisms given by the Bell/LaPadula and Biba policies and an application authentication function. The functions must be implemented by the operating system. An abstract model of the smart card operating system defines data types used to define system states and the operating system commands.

It has been verified that the abstract system model satisfies the security objectives.

However, the model considers communication through storage channels between applications that are stored on the card. The responsibility of the secure communication between on-card applications and external applications or external devices is put on the applications running on the card.

4.4 BasicCard

The medium-sized German company ZeitControl launched in 2004 a third version of their smart card operating system ZC6.5 called MultiApplication BasicCard. Applications are implemented in the ZC-Basic programming language, which is the Basic programming language enhanced with special features for the processor card. The ZC-Basic source code is compiled into P-Code and can be divided into a Terminal and Card program. The former is interpreted by a MS Windows program and the latter is loaded onto the card and interpreted by it. The smart card operating system provides mechanisms for the secure download of new BasicCard applications, for instance the card can be configured in a way that it accepts digitally signed applications of the card provider only. Moreover, it considers attributes on files allowing either the sharing of the file between a specific group of applications or the hiding of the file to other applications on the card similar to Java Cards.

Except information given by data sheets of ZeitControl there is no analysis of the security functions known. As in all other smart cards the BasicCard additionally provides a library of cryptographic algorithms that can be used to secure the communication between an on-card application and the terminal program, whereby the functions must be integrated in the applications. They are not forced by the operating system or interpreter.

4.5 Limitations of the security policies

All smart card operating systems introduced in the previous sections provide a multi-application platform and allow the download of new applications. They strongly differ in the implemented security policy. We can say the basic Java Card, i.e. without Global Platform features, is operated in an user oriented way, because it is controlled by the user whether an applet should be loaded or not. The JVM verifies the electronic signature of the applet but these signature simply signals that the applet has been compiled with

	Java Card	**Multos**	**SMaCOS**	**Basic Card**
`SO1`: secure download of applications	■	■	■	■
`SO2/SO3`: application isolation	■	■	■	■
`SO4`: communication within in the card	■		■	■
`SO5/SO6`: external communication	■			

■ fully supported, □ not supported

Table 4.1: Comparison of multi-applicative smart card operating systems

a valid converter. It is in the responsibility of the user if the functionality of an applet will interfere the still loaded applets on the card. With the introduction of the Card Manager a Java Card can be operated in an issuer oriented way. Applets assigned to different contexts may totally be isolated on the card; but if one of the applets grants access to one of their methods we can not say the isolation is total. Additionally, there is no mechanism that let an applet trust a method of another applet.

Because the Multos security policy does not allow communication between applications at all, this is a total isolation. In addition, before a new application may be loaded onto the card it must be evaluated by the issuer; the card does not accept any other applications. Hence, a Multos card is operated in an issuer oriented way. It can be enriched by a trusted third party that does the evaluation as a service provider.

A second issuer oriented security policy is implemented by SMaCOS. It classifies every application in confidentiality as well as in integrity classes. The integrity level may be determined by the Evaluation Assurance Level under the Common Criteria or ITSEC scheme. Hence, the evaluation is done by third parties. Based on the classification an application gets well defined access rights in order to communicate with other applications. If an application does not want any communication it runs absolutely isolated on the card. Furthermore, the application are classified before they can be loaded onto the card. The issuer determines the confidentiality class and thus, no application can be loaded onto the card without the issuer's consent.

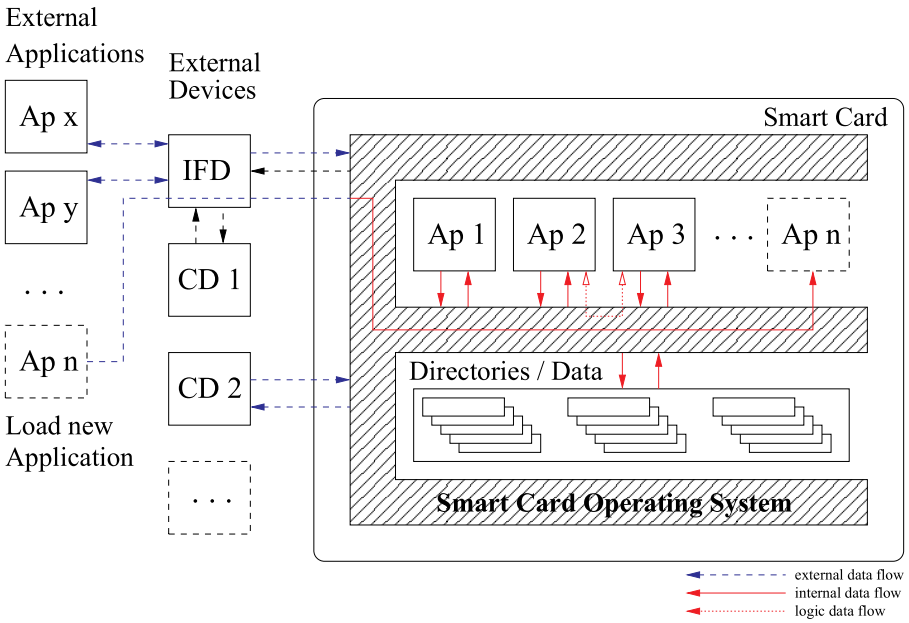


Figure 4.4: An extended smart card operating system design

The Basic Card provides a service to verify electronic signatures of applications to be loaded onto the card. The validity of a certain electronic signature is set by the card issuer or the user and thus depends on the initial configuration of the card. It additionally provides services for application isolation and inter-application communication, but there is no detailed description or analysis of the mechanisms published.

None of the introduced operating systems consider the communication to applications outside the card. There are techniques proposed that enable Java Card applets to gain access rights through electronic tickets that hold special capabilities. These techniques can easily be adapted by other smart card security policies because they are application driven. In Table 4.1 a summary and comparison of the operating systems is given regarding the implemented security policy.

As shown in Table 4.1 there is a lack in integrating external applications or devices in the smart card operating system. We want the operating system to do both act as a firewall between on-card applications and act as a firewall between an on-card application and external applications or devices. This may be demanded by applications as mentioned in the example `BioSig`. In the example the execution of the `PIN` application is only allowed if a smart card terminal with integrated PIN pad is used. Furthermore, the `Bio1` application is only allowed to accept biometric data from an `EAL4high` evaluated capture device. We want the operating system to check whether

an external device or external application is allowed to communicate with a certain on-card application or not.

That means an application as well as the external devices, e.g. a smart card terminal or a biometric capture device, first have to be authenticated by the smart card operating system. In accordance to the security policy the operating system then classifies the applications and devices and therefore activates all the applications on the card. In this way the communication channels are established. To follow the `BioSig` example, an external signature application running on a host computer that uses a very simple smart card terminal, i.e. no integrated PIN pad, is not allowed to execute the `PIN` application even if the application is sufficiently evaluated. To make it more precise, the evaluated application may execute the `QeSig` application but `QeSig` can not be activated via the `PIN` application. If a sufficiently evaluated biometric capture device is authenticated by the smart card the `QeSig` application could be activated via `Bio1`.

To summarize, we want to move the control of allowed communications between applications on the card and applications outside the card from the applications themselves to the operating system. This is very helpful in issuer oriented operating systems, because it lightens the application development and the application evaluation process. If an operating system with such functionalities is sufficiently evaluated all applications can take advantage of those functions and thus, are easier to evaluate; this may also reduce development costs. That is why we are concerned with an operating system design that takes those functionalities into account.

An overall design of such system is illustrated in Figure 4.4. The dashed blue colored arrows determine the external data flow and the full red colored arrows determine the internal data flow. Generally, confidentiality and integrity requirements between applications on the card can be achieved by access control mechanisms of the operating system because those mechanisms are under the full control of the operating system. Hence, information flow between applications on the card is achievable by access control mechanisms, which is illustrated with red dotted arrows in the figure. As the communication between an external application and an application on the card is via an open network, confidentiality and integrity requirements can not be fulfilled by access control mechanisms. Other mechanisms have to be taken into account as they are given by cryptographic protocols. In the next chapter we are going to develop the security functions for the internal and external communication in more detail considering the SMaCOS design.

Chapter 5

The extended security policy

The extended security policy and its model proposed in this chapter is based on the SMaCOS security model. We first give a detailed introduction of the SMaCOS security policy and its model in order to extend it with an execution right and to integrate external applications and devices. Then, we informally show that the defined security functions implement the security objectives discussed in Section 3.3 and counter the security threats identified in Section 3.2. To demonstrate the feasibility of the developed extended security policy we apply the `BioSig` example introduced in Section 2.3.

5.1 The SMaCOS security model

The SMaCOS security policy model combines the secrecy policy model by Bell and LaPadula (BLP) (BL76) and the integrity policy model by Biba (Bib77), which defines a multi level security system (MLS). Generally, MLS operating systems implement a reference monitor that supervises all operating system calls (or commands) and checks the access conditions to decide whether the command is authorized to be executed or not. Hence, the reference monitor controls the access rights of applications or processes to data or other resources according to rules given by the security policy.

The SMaCOS security policy model defines partial ordered *access classes (or level)* $SecL$, $IntL$ and *access categories* $SecC$, $IntC$ for secrecy and integrity, respectively. Each subject $s_i \in S$ and object $o_j \in O$ has to be assigned to a specific access class in a specific access category via the assignment functions S_{cls} and I_{cls} for secrecy and integrity. In addition, every access category defines its own access classes, which is given by the assignment functions S_{ctg} and I_{ctg}.

If a subject wishes to gain read access to an object, the access class of

subjects objects	$S = \{s_1, \ldots, s_a\}$ $O = \{o_1, \ldots, o_b\}$	
	secrecy	integrity
access category	$SecC = \{sc_1, \ldots, sc_c\}$	$IntC = \{ic_1, \ldots, ic_e\}$
access class, $p \in \{r, w, x\}$	$SecL = \{sl^{sc_k}_{\{p\}1}, \ldots, sl^{sc_k}_{\{p\}d}\}$	$IntL = \{il^{ic_l}_{\{p\}1}, \ldots, il^{ic_l}_{\{p\}f}\}$
category assignment fct	$S_{ctg} : SecL \rightarrow SecC$	$I_{ctg} : IntL \rightarrow IntC$
class assignment function	$S_{cls} : S \cup O \rightarrow SecL^{SecC}$	$I_{cls} : S \cup O \rightarrow IntL^{IntC}$

Table 5.1: Elements of the SMaCOS policy model

the object must be less than or equal to the access class of the subject. This rule is called *simple security property*. It simply hinders subjects from reading higher classified objects. The second secrecy rule is the **-property* (confinement property), which states that the access class of the subject must be less than or equal to the access class of the object in order to get write access to the object. In other words, a subject that is cleared for the secrecy access class `secret` in category `X` is not allowed to read an object of access class `top secret` in category `X` and is additionally not allowed to read an object of access class `secret` in category `Y`, if we assume `top secret` greater than `secret`.

The integrity model defines similar *integrity access classes* and *integrity access categories* as well as the *simple integrity property* and the *integrity confinement property*. They state that the access class of a subject must be less than or equal to the access class of the object in order to get read access and vice versa to get write access. It prevents applications of high integrity of reading data or executing programs of low integrity. The elements of both models are summarized in Table 5.1. The set $\{p\}$ denotes the *read, write* and *execute* access class.

The read, write and execute permissions can be expressed as follows:

$$\begin{aligned} read(s_i, o_j) \iff & \; sl^{sc_k}_r(s_i) \geq sl^{sc_k}_r(o_j) \; \wedge \\ & \; il^{ic_l}_r(s_i) \leq il^{ic_l}_r(o_j) \end{aligned} \tag{5.1}$$

Intuitively, equation 5.1 states that an arbitrary subject s_i, e.g. an application, with read secrecy access right $sl^{sc_k}_r$ in secrecy access category sc_k and with read integrity access right $il^{ic_l}_r$ in integrity access category ic_l get the permission of reading an arbitrary object o_j if and only if the read secrecy access right is greater than or equal to the read secrecy access right of the object and if the read integrity access right is less than or equal to the read integrity access right of the object. The equations 5.2 and 5.3 give the write permission of an arbitrary subject s_i to an arbitrary object o_j and the execute

permission of an arbitrary subject s_i to an arbitrary subject s_j.

$$write(s_i, o_j) \iff \begin{array}{rcl} sl_w^{sc_k}(s_i) & \leq & sl_w^{sc_k}(o_j) \\ il_w^{ic_l}(s_i) & \geq & il_w^{ic_l}(o_j) \end{array} \wedge \tag{5.2}$$

$$exec(s_i, s_j) \iff \begin{array}{rcl} sl_x^{sc_k}(s_i) & \geq & sl_x^{sc_k}(s_j) \\ il_x^{ic_l}(s_i) & \leq & il_x^{ic_l}(s_j) \end{array} \wedge \tag{5.3}$$

All three rules must hold at the same time in order to define a *secure* system. If we look at the system as a state machine it must be secure in the initial state and in all reachable states. The state transitions are defined by all possible system operations. It must be guaranteed that every system operation respects the access rules.

In our model we distinguish the *read* and *write* operation from the *execute* operation. Figure 5.1 illustrates the resulting access rights when assigning applications `Ap1` and `Ap2` to secrecy access class `high` and integrity access classes `low` of a single access category `X`. Application `Ap3` is assigned to secrecy access class `low` and integrity access classes `high`, whereby it holds the relation `low` < `high`. According to the rules application `Ap1` is cleared for reading and executing applications `Ap2` and `Ap3` but is not allowed to write data to application `Ap3`. We work out the more illustrative example `BioSig` in Section 5.4.

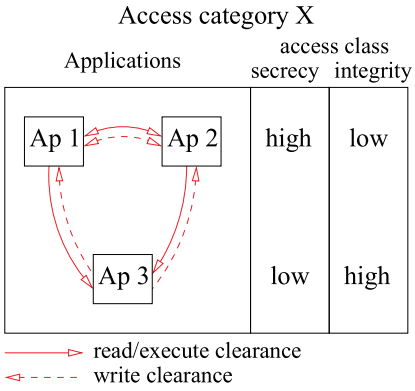


Figure 5.1: Access rights between applications in SMaCOS

The main problem of the classical approach is the incompatibility of the BLP model with real world systems if the alteration of the access classes at runtime is prohibited at all, because this could lead to the highest classification of all objects and even subjects. On the other hand, if every subject is allowed to freely alter the classification of the objects the system would not fulfill the intended security policy, because every subject could classify

the objects to the lowest level. A generic framework of *secure transitions* marking subject and objects up or down is given in (McL94b).

Furthermore, *trusted subjects* have been introduced that are not constrained by the *-property. Hence, those subjects are allowed to downgrade the classifications of objects in the BLP model. In our model we treat applications as special objects, e.g. executable files. The assignment of those objects to the *exec access class* is done by the smart card operating system in the loading process. Once an application is loaded the classification is not alterable. In contrast, we allow the applications to set the access classes of objects they created, which may be the same or a higher secrecy level in accordance to the above mentioned rules. That is, let an application be assigned to the write access classes $sl_w^{sc_k}$ and $il_w^{ic_l}$. A new file created by the application can now be assigned by it to the same write access classes $sl_w^{sc_k}$ and $il_w^{ic_l}$ or to all secrecy write access classes higher than $sl_w^{sc_k}$ or to all integrity access classes lower than $il_w^{ic_l}$ in the same access category. Hence, in our model we are not confronted with the above mentioned problem of downgrading and upgrading of objects.

The SMaCOS model does not take an execution right into account. A similar model of security policy given in (KAT00a) distinguishes two kinds of executive permissions: a normal *transfer* and a special *CHAIN* operation. The transfer rule states that the integrity level of the process must be equal to or less than the integrity level of the executable object. The new program runs at the level of the calling process. The CHAIN operation is independent of the calling process and states that an executable object can always be executed and runs at its predefined level.

The operation in our model given in Equation 5.3 defines a combination of both, because a subject s_i has clearance to execute subject s_j if and only if the secrecy access level of s_i is greater than or equal to the secrecy access level of s_j and reverse for the integrity level. The rule is similar to the read rule 5.1 if we see the subject s_j as an executable object. In other words an application of a high integrity level is not allowed to execute an application of low integrity level. In addition, an application of low secrecy level is not allowed to execute an application of a high secrecy level, because confidential data could be revealed. After an application is executed it runs at the access classes defined by the card issuer and set by the smart card operating system.

5.1.1 Communication channels

Communication can be realized via storage channels, which means the return values are stored in a temporary file. Hence, the called application needs to have write permission to the temporary file and the calling application needs

to have read permission to the file. This is in fact true, because otherwise the calling application could not have executed the called application. In the SMaCOS model the calling of an application is seen as not security relevant. In our extended model we are heavily concerned with this issue and give an example in the next section. We already defined the allowed communication channels within the model according to the access rules. It turns out, if two applications `Ap1` and `Ap2` are assigned to two different access categories they are completely separated on the card. But it is often the intention to allow a secure communication between those applications.

There are three main techniques in achieving this. Firstly, objects, e.g. data files, are multiple assigned to both access categories of `Ap1` and `Ap2`. So, both applications have access to the object. This also works vices versa, here subjects, e.g. the applications, are multiple assigned to the different access categories and all objects belong to one of the access categories. Thirdly, one can define *trusted channel programs* that have two different pairs of access levels in different access categories. One pair is used for reading and needs to be assigned to the access category and access class of the first application `Ap1`. The second pair has write clearance of the second application `Ap2`. Hence, the channel program can read the content of a file of application `Ap1` and write it to the second application `Ap2`.

Those channel programs can also be used within a single access category for downgrading or upgrading of data to a lower or higher access class. This is also known as *sanitization*. But special care has to be taken in introducing such channel programs because they intentionally violate the overall security policy. On the other hand they are a powerful tool for establishing special communication channels. Especially virus checker applications or firewall applications take advantage of it. In SMaCOS the channel programs are modeled as special applications that can be loaded onto the card. To make sure that only approved channel programs are loaded onto the card it has to be authenticated by both communication partners. Hence, in the model an application is be seen as a set of programs and channel programs. After the successful and secure loading of an application onto the card it is assigned to its own access category by the operating system. The application is now allowed to freely download programs and to create files and to assign the files to access classes within the access category according to the access rules. Because of the separate access category it can not interfere with other applications unless it agrees to a channel program. The *transfer* rule can be expressed as follows:

$$\begin{array}{rcrcl} transfer(s_i, s_j) & \Longleftrightarrow & sl_w^{sc_k}(s_i) & \leq & sl_r^{sc_l}(s_j) \quad \wedge \\ & & il_w^{ic_k}(s_i) & \geq & il_r^{ic_l}(s_j) \end{array} \tag{5.4}$$

Intuitively Equation 5.4 states that a channel program is allowed to transfer information from an arbitrary subject s_i to an arbitrary subject s_j if and only if the write secrecy class of subject s_i in secrecy category sc_k is equal to or less than the read secrecy class of subject s_j in secrecy category sc_k and the write integrity class of subject s_i in integrity category sc_k is equal to or greater than the read integrity class of subject s_j in integrity category sc_k. Please note, the access categories for subject s_i and s_j may differ, which leads to the problem that there is no ordering between the access classes[1]. Before a channel program can be used the ordering must be defined.

In other words the channel program is assigned to two pairs of access classes for reading and writing as well as for secrecy and integrity, respectively. The pair used for reading will have the clearance of subject s_i, while the one used for writing will have the clearance of subject s_j. This allows the channel program to read the content of a file from s_i and write it into a file that can be read by s_j.

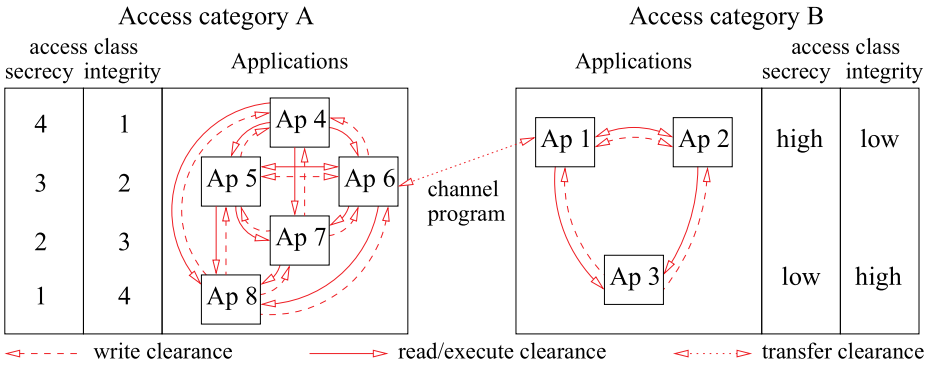


Figure 5.2: Communication channels between applications

A summary of the allowed communication channels between applications assigned to different access categories and access classes is given in Figure 5.2. In the figure the idealized application `Ap6` is assigned to secrecy access class 3 and integrity access class 2 in access category `A`. We assume access class 4_A of category A is greater than 3_A and so forth. According to the access rules it is cleared to read objects of applications `Ap5, Ap7` and `Ap8` as well as to execute the applications. It has also write permission of objects of applications `Ap4` and `Ap5`. According to the assignments there is no rule that allows communication between applications assigned to different access categories. Nevertheless, we want allow a communication between applica-

[1]The BLP and Biba models require a partial ordering of the access classes, which is two access classes of the same access category are in order but there is no ordering of two access classes of different access categories.

tions `Ap6` and `Ap1`, which can be realized with an additional channel program. Therefore, the ordering between the access classes of category A and B must be defined, which is $3_A = high$ and $2_A > low$ in the example.

A further possibility of creating a communication between applications `Ap6` and `Ap1` is to additionally assign application `Ap6` to access levels of access category `B` as shown in Figure 5.3. Please note, when applying this mechanism there are more than the intended communications between `Ap6` and `Ap1` implicitly defined.

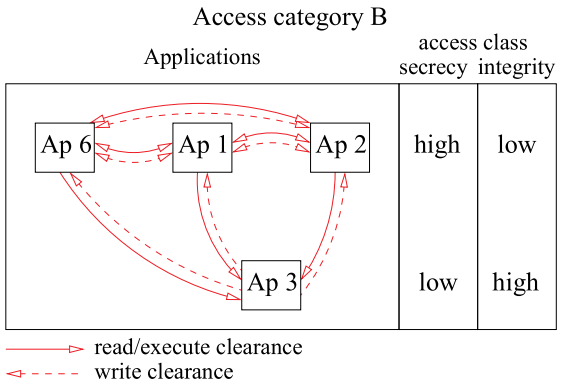


Figure 5.3: Communication channels between applications

While the secrecy model is based on the military security system with access levels like $SecL = \{unclassified, confidential, secret, top\text{-}secret\}$ and access categories like $SecC = \{navy, army, air\text{-}force\}$ there was no practical application for the integrity model. The question to be answered was, which program is of higher integrity than another? One first answer is given in (KAT00b), which uses the evaluation levels in terms of the ITSEC. They say that an application evaluated with `E4` is of higher integrity than an application evaluated with `E2`, which is $IntL = \{\texttt{E1}, \texttt{E2}, \texttt{E3}, \texttt{E4}, \texttt{E5}, \texttt{E6}\}$. But in their approach they do not address integrity categories. In our extended model we will take advantage of categories in order to manage different evaluation schemes and to implement legal requirements in the example scenario `BioSig`.

5.1.2 Loading new applications

Bearing in mind the security objectives mentioned in Section 3.3 on Page 23 we did not say anything about how the applications (the subjects) are assigned to security levels within the card. There is always one point in time, where every application must be loaded onto the card and hence, will be assigned to the defined access level. It is very crucial for the whole system

that this initial process is done in a secure way. Imagine a malicious application that is somehow loaded onto the card and assigns itself to the highest secrecy and integrity access class for all categories. This application would be allowed to read all data and to distribute (malicious) data to all other applications.

To prevent the card of loading such applications we again follow the approach given in (SRS+00). Every card holds a trusted key that is used to verify electronic signatures. In an off card process every application together with assignment information is electronically signed by the issuer of the specific card. Therefore, a card only accepts applications that provide a valid electronic signature and hence assigns the stated access levels to the application. All other applications are denied.

5.2 External applications and devices

The SMaCOS security policy and its model describe rules that define access rights of applications to data or other applications. These rules shall be enforced by operating system functions. Moreover, the rules address the communication between on-card applications and the secure download of new applications only. The aim of this section is the integration of external applications and devices in a way that the given rules need not to be changed. To express the mechanism we first give the basic idea and then discuss a simple example taken from electronic signatures.

5.2.1 The basic idea

We again consider the application scenario illustrated in Figure 5.3. The figure shows all communication channels between applications `Ap1`, `Ap2`, `Ap3` and `Ap6` according to the access rules 5.1, 5.2, 5.3 and 5.4, whereby `Ap6` has been double assigned in order to allow a communication between `Ap6` and `Ap1`. A second alternative in achieving this is the definition of a channel program, which is not considered here. In the described scenario application `Ap6` is an on-card application and all communications are controlled by the operating system.

In a modified scenario we assume application `Ap6` to be an external application, for example running on a separate smart card. We again want a communication between application `Ap6` and `Ap1` in accordance to the access rules. To achieve this we create a *dummy application* `[Ap*]` on the smart card and assign the dummy to the intended secrecy and integrity level. The square brackets indicate such a dummy application that can be instantiated

with a proper external application. Hence, the square brackets additionally define requirements for an application to be assigned to the dummy application. If an external application `Ap6` wants to communicate with application `Ap1` it has to be assigned to the dummy application `[Ap*]` after a successful check for appropriateness. The curly brackets denote an *application set* because there is usually more than one external application matching the `[Ap*]` requirements. The modified scenario is illustrated in Figure 5.4.

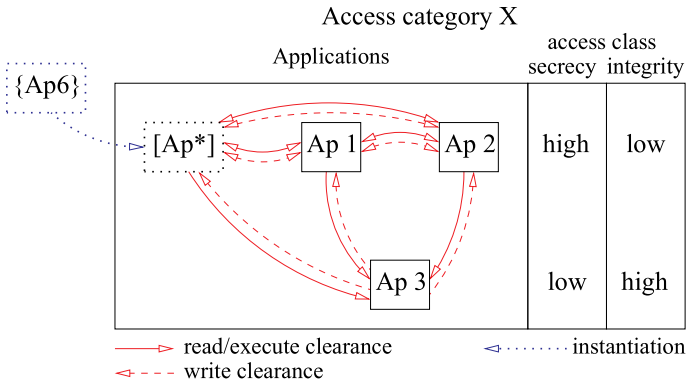


Figure 5.4: Integration of external applications

In the figure the red full and dashed arrows denote internal communication channels, whereas the blue dotted lines denote external communications. Once an application has been assigned to a dummy application it is handled like an internal application and all access rules are applied. For instance, from the point of view of application `Ap1` there is no difference between `Ap6` and `Ap2`. Since the red colored communications (full and dashed arrows) are under the full control of the operating system this is not true for the blue colored communication (dotted lines), which is over an open network. Confidentiality or integrity requirements must be realized with other means than access control. Before we discuss this in more detail in Section 5.2.3 we illustrate the mechanism again by considering a simple example in the next section.

5.2.2 Electronic signatures

We work out a simplified scenario `eSig` of the discussed `BioSig` example in Section 2.3 on Page 10. Therefore, a smart card is considered holding two signature creation applications `QeSig` and `AeSig`, whereby the former is assumed to create a qualified signature and the latter is assumed to be a PGP signature creation application. According to legal regulations we assign `QeSig` to a high integrity level of an access category `eSig` and `AeSig` to a low

integrity level. Both application get as input a hash value of a document to be signed with a secret cryptographic key. The signature creation has to be performed after the successful authentication of the corresponding user only. Since the hash value must not be handled confidentially but full of integrity, we are not concerned with secrecy access levels here.

Therefore, we assume two user authentication applications on the card; `PIN` is an application asking the user for a personal identification number and `Bio` is a biometric authentication application. Because the `PIN` application provides a higher strength of function than the `Bio` application we assign `PIN` to the high integrity level and `Bio` to the low integrity level. That might look strange but in fact, biometric authentication techniques can not provide the high strength of function as PIN based methods can achieve (with three authentication attempts only) due to the performance measurement in terms of FAR and FRR (see Section 2.3.2).

The `QeSig` application is required to use the `PIN` application only, whereby `AeSig` is free in choosing either the `PIN` or `Bio` authentication. The defined assignments illustrated in Figure 5.5 fulfill this requirement. So far, we modeled internal communications only.

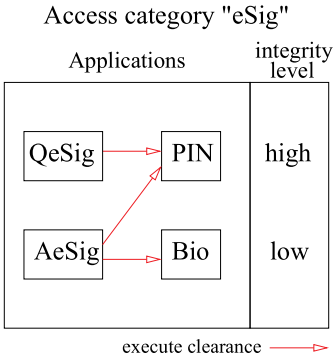


Figure 5.5: Electronic signature example

As above mentioned the applications `QeSig` and `AeSig` expect a hash value as input. The hash values of documents have to be created by external applications, usually word processing programs. We again distinguish trustworthy word processing programs that may come with a trusted viewer and others. We want those trusted external signature applications `SigAp` to access the trusted signature creation application `QeSig` only. Therefore, we define two dummy applications `SigHigh` and `SigLow` on the card that are assigned to the high and low integrity level, respectively.

If a trusted external signature application `SigAp` contacts the smart card in order to create an electronic signature it is assigned to the `SigHigh` dummy

application and hence, is only allowed to execute the `QeSig` application. This design makes it necessary that `SigAp` has no chance to alter its own identity, which can in fact be assumed for evaluated signature applications according to legal requirements. All other external signature applications are assigned to `SigLow` and hence, are free in choosing the signature creation application. The described scenario is illustrated in Figure 5.6.

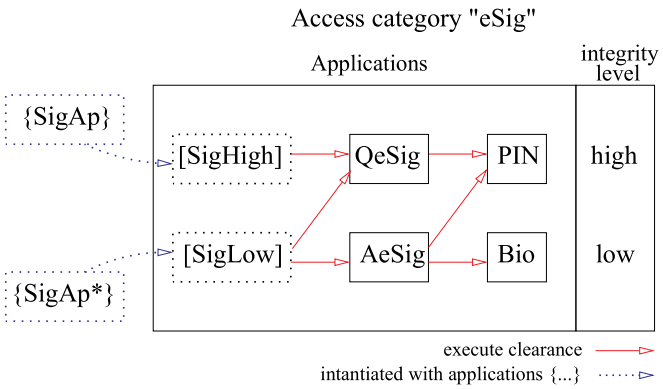


Figure 5.6: Integration of external signature applications

The overall aim of the described scenario is to keep a level of trust. A trusted (and evaluated) external signature application `SigAp` shall have no chance to choose a signature creation application other than the trusted `QeSig` that again can only choose a trusted authentication application like the `PIN` application. Moreover, the trusted authentication application `PIN` shall be able to choose a trusted PIN pad only, which is not displayed in the figure. As already mentioned, different security functions have to be used in order to achieve integrity of the transferred data between `SigAp` and `SigHigh` and between `SigHigh` and `QeSig`, which is discussed in the next section. The fully worked out case study `BioSig` is given in Section 5.4.

5.2.3 The extended security model

In our extended model subjects are related to applications on the card as well as outside the card. Because the operating system of the card has full control of both all on-card applications and data stored on the card, this does not hold for external applications. Those communications are over an open network, which makes the additional protection of the communication channel between external and internal (dummy) applications necessary. Furthermore, the external applications have to be authenticated by the card every time they wish to communicate with an internal application in order to determine the dummy application they belong to.

In contrast to the procedure of loading internal applications onto the card mentioned in Section 5.1.2, the card first has to authenticate the application and its assignment information, then it has to assign the application to the defined dummy application and finally may has to establish a session key in order to guaranty confidentiality or integrity of the transferred data. To achieve this the external application may present a certificate, e.g. in card verifiable format or in X.509 format, that holds the public key together with the assignment information. The application should hold the corresponding private key. Hence, the card can verify the certificate by using the appropriate trusted public key stored on the card. Here, security depends on the chosen authentication mechanism and the key establishment protocol. Well understood and standardized cryptographic protocols are given in ISO/IEC 9798. Additionally, a mutual authentication may be performed if requested by the external application. After this procedure the external application can be treated as an internal application.

The blue dotted lines given in Figure 5.4 and Figure 5.6 denote a secure tunnel. In the following we put the extended SMaCOS access rules 5.1, 5.2, 5.3 and 5.4 in an Access Control Component `AccessCtrl` of a smart card model. To provide a secure tunnel between component `AccessCtrl` and the external applications we need to extend the smart card model with an additional gateway component that manages all external communications. The new communication component `CommC` controls all the input and output traffic of the entire smart card as illustrated in Figure 5.7. There is no other input/output channel.

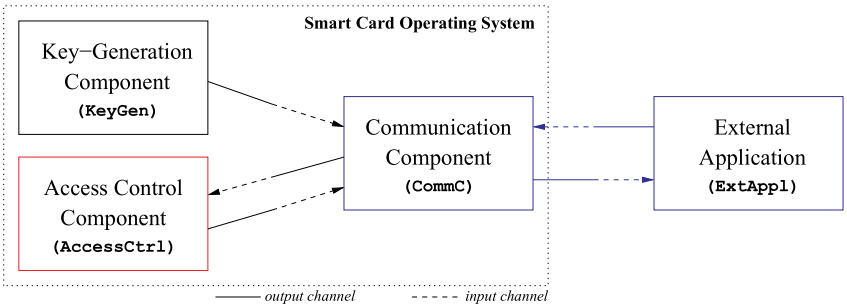


Figure 5.7: The extended smart card model

For instance, the communication component accepts a communication request by an external application and authenticates the application in order to assign it to the specific dummy application. If demanded by one of the applications the communication component provides a session key in order to achieve confidentiality or integrity of the transferred data. Therefore, a further component `KeyGen` is added that generates session keys in advance

and stores them in an internal buffer. The introduction of this component is due to the fact that the generation of keys or nonces may take time. The simplified design of the extended smart card model is illustrated in Figure 5.7. We refine and verify the smart card model in Chapter 7.

5.3 Summary of security functions

In Section 3.3 we discussed the security objectives that the smart card operating system should fulfill in order to counter the potential security threats mentioned in Section 3.2. In the following we are going to summarize the security functions introduced in the previous section and show how the certain functions implement the security objectives.

The combined Bell/LaPadula and Biba model is expressed by the three rules for reading, writing and executing an object by a subject given by the access rules 5.1, 5.2 and 5.3. These rules define security enforcing functions. In addition we allow special communication channels implemented by trusted channel programs and defined by rule 5.4 as suggested by (SRSS99). We name the security functions as follows:

`SF1`: No read up for secrecy and no read down for integrity (Rule 5.1);

`SF2`: No write down for secrecy and no write up for integrity (Rule 5.2);

`SF3`: No execution upwards for secrecy and no execution downwards for integrity (Rule 5.3);

`SF4`: Channel program (combined read and write clearance)(Rule 5.4).

The mechanisms given by the Bell/LaPadula and Biba model represented by the security functions `SF1`, `SF2`,`SF3` and `SF4` as well as the proper definition and the proper assignment of access classes implement the security objective `SO4` and counter the security threats `ST7` and `ST9`. Regarding the threat `ST9` we will give additional explanations below.

The security objective `SO2` can be implemented in a similar way. Therefore we define two special system access categories $SecC = IntC = \{sys\}$ and access classes $SecL = IntL = \{low_{\{p\}}\}$ for secrecy and integrity, respectively. Furthermore, we define $S_{ctg}(low_{\{p\}}) = sys$ and $I_{ctg}(low_{\{p\}}) = sys$. The operating system *os* itself is assigned to the read, write and execution access classes $S_{cls}(os) = low^{sys}_{\{rwx\}}$ and $I_{cls}(os) = low^{sys}_{\{rwx\}}$. All other applications app_i are assigned to the execution access classes $S_{cls}(app_i) = low^{sys}_{x}$ and $I_{cls}(app_i) = low^{sys}_{x}$, only. Because of this default setting by the system no application is allowed to read or write other applications executable file.

Security threats	Security objectives					
	SO1	SO2	SO3	SO4	SO5	SO6
ST1	SF5					
ST2						SF6
ST3					SF6	SF6
ST4					SF6	
ST5	SF5					
ST6	SF5					
ST7		SF1-SF4		SF1-SF4		
ST8			SF1-SF4			
ST9				SF1-SF4		
ST10	not under consideration					
ST11						

Table 5.2: Security functions implementing the security objectives and countering the security threats

Note, that in this case we treat the application as an object, e.g. the executable file. So again, the security functions SF1-SF4 implement objective SO2 and counter the threat ST7.

With the successful authentication of every application that is loaded onto the card our model matches the security objective SO1. In addition the authentication includes the authentication of the assignment information given by the card issuer. So, it is up to an off card process to define various access classes according to very different business cases. Please note, that the assigned access classes of an application determine the communication channels between all applications, especially if the application acts as a channel program. We will explain this in more detail in the next section using the BioSig example.

The authentication of loaded applications and assignment information is

a further security function:

SF5: Authentication of loaded applications and assignment information.

The function SF5 implements the objective SO1 and hence counters threat ST1 as well as threats ST5 and ST6 because only successful authenticated applications and its identities can be loaded onto the card. Note, it is the operating system that presents the identity of an application to external applications or devices.

The security function SF6 defines the use of cryptographic protocols in order to establish an authentic and confidential communication between external applications or devices and the smart card applications. The cryptographic protocols in question must provide functionalities for authenticating the communication partners as well as for establishing a session key in order to encrypt the transferred data. The key can also be used to guaranty integrity of the transferred data, for instance by means of message authentication codes. To do so, we use the assumption that the external applications and external devices are evaluated under a specific scheme and furthermore are equipped with an electronic certificate that state both the validity of the public key and the evaluation level and scheme. One may say this is a very strong assumption. This is partly true. In fact most of the security hardware and software is already evaluated (mostly under the CC) but the evaluation bodies generally issue a certificate in paper form that can not be used here. In our scenario the evaluation body should create a key pair and a corresponding certificate, e.g. using a PKI, and should securely store the private key in the application or device in question. This of course can only be done hand in hand with the developers. We summarize security function SF6:

SF6: Authentication of external applications and devices and assignment information as well as confidentiality and integrity of transferred data.

Hence, the function SF6 implements both security objective SO5 and SO6 and counters security threats ST2, ST3 and ST4.

Table 5.2 summarizes the relations between the security functions, the security objectives and the security threats. It turns out that we did not develop security functions countering the threats ST10 and ST11. The former threat addresses the problem of *covert channels*, which may allow information flow even the channel is not designed for communication at all. As an example one may think of a *high* classified process A and a *low* classified process B that somehow share the same resource, e.g. a display. Process A may access the display several times using a specific time delay that may be interpreted

by process B. Thus, information flows from A to B which is not intended. There are more examples given in (And01). Those kinds of covered channels are often referred to as *timing channels* in contrast to *storage channels*. A very simple example of a covered storage channel is a process A that creates a file of a specific size while the reading of the file by process B may be prohibited. Let listing the file and getting the size information be allowed the processes could abuse the covered channel for information flow. In this manner the security functions `SF1-SF4` can prevent storage channels. Timing channels are very hard to handle and require further mechanisms to put into account as resource separation or individual CPU clock cycles. That is why security functions countering threat `ST10` are not within our security policy model.

Security threat `ST11` addresses information that are transmitted via a legal channel between applications, whereby the information are not of those kind as intended by the purpose of the application. In our security policy model we put the responsibility of the handling of information on the applications themselves. One can think of enriching the operating system functionality with a plausibility check of transmitted data via a legal communication channel, e.g. the input of a signature creation application is always a hash value. This makes a very precise definition of *input* and *output* data for every single application necessary. We do not take such functionality into account and therefore do not provide security functions to counter security threat `ST11`.

5.4 Application to the BioSig example

In this section we demonstrate how a security policy given by an overall application can be implemented using the proposed extended security model of the smart card. In Section 2.3 we defined two signature creation applications `AeSig` and `QeSig` that can be activated by various authentication applications `PID`, `PWD`, `Bio1` and `Bio2`. Furthermore, an external signature application running on a host computer may execute one of the signature creation applications running on the smart card. The access rights of all applications in the example 2.3 are given in the activation matrix in Table 2.1 and in Figure 2.2.

In our design we do not make use of channel programs as described in Section 5.1.1. We rather choose the second technique for establishing the communication channels between applications, which is the multiple assigning of subjects to access categories. We give an alternative design that uses channels programs in Section 5.5.

Firstly, we define two integrity access categories $IntC = \{biosig, auth\}$ as well as integrity access classes

$$IntL = \{low_{\{p\}}, med_{\{p\}}, high_{\{p\}}\}$$

and assign the classes to the categories

$$I_{ctg}(low_{\{p\}}) = I_{ctg}(high_{\{p\}}) = biosig$$
$$I_{ctg}(low_{\{p\}}) = I_{ctg}(med_{\{p\}}) = I_{ctg}(high_{\{p\}}) = auth\ .$$

Please note, by default there are also $SecC = IntC = \{sys\}$ and the corresponding assignments defined. Because of simplicity we are not going to mention it here. The set $\{p\}$ denotes the *read, write* and *execute* access class denoted as r, w, x, respectively. In the `BioSig` example we only consider $p = \{x\}$, because we are mainly concerned with the execution of other applications. In fact, once an application is executed the calling application usually expects a result value. This kind of communication is realized via file sharing, so the called application needs to have write permission to a temporary file and the calling application needs to have read permission to the file.

Next, we have to assign all applications to the access classes as shown in Table 5.3. The application `AeSig` is assigned to access class $low_{\{x\}}^{biosig}$ and `QeSig` is assigned to $high_{\{x\}}^{biosig}$. If an external signature application `ExtSig` wants to execute the signature creation application `QeSig` it must be cleared for integrity access class $high_{\{x\}}^{biosig}$. Therefore, we have to define a dummy application and assign it to the access class $high_{\{x\}}^{biosig}$, which is denoted as $I_{cls}([\texttt{EAL4high}]) = high_{\{x\}}^{biosig}$. The term [`EAL4high`] represents the dummy application that in our case requires an `EAL4high` evaluation under the CC scheme for the signature application.

Moreover, for flexibility we assign all higher evaluation levels also to the $high_{\{x\}}^{biosig}$ access class, which are [`EAL5high`], [`EAL6high`], [`EAL7high`]. One could also define dummy applications for the ITSEC scheme, which would make the assignments [`E3high`], [`E4high`], [`E5high`], [`E6high`] to $high_{\{x\}}^{biosig}$ necessary. But in the following we will concentrate on the Common Criteria only. At least, one could bundle all evaluation levels in a single dummy application to keep the number of dummy applications small. Please note, we can do those assignments in our `BioSig` example, because we want allow even higher evaluated external signature applications to use the `QeSig` application. This is due to legal requirements and can not be adapted by other scenarios. Our aim is to prevent high evaluated external signature applications of using the not evaluated `AeSig` application.

Once the smart card successfully authenticated the external signature application `ExtSig`, which means the smart card is convinced of the evaluation level, it assigns the application to the defined access classes [`EAL4high`] =

$I_{cls}(\texttt{AeSig})$	$=$	$low^{biosig}_{\{x\}}$	$I_{cls}(\texttt{AeSig})$	$=$	$low^{auth}_{\{x\}}$
$I_{cls}(\texttt{[no-eval]})$	$=$	$low^{biosig}_{\{x\}}$	$I_{cls}(\texttt{PWD})$	$=$	$low^{auth}_{\{x\}}$
$I_{cls}(\texttt{QeSig})$	$=$	$high^{biosig}_{\{x\}}$	$I_{cls}(\texttt{QeSig})$	$=$	$high^{auth}_{\{x\}}$
$I_{cls}(\texttt{[EAL4high]})$	$=$	$high^{biosig}_{\{x\}}$	$I_{cls}(\texttt{PIN})$	$=$	$high^{auth}_{\{x\}}$
$I_{cls}(\texttt{[EAL5high]})$	$=$	$high^{biosig}_{\{x\}}$	$I_{cls}(\texttt{Bio1})$	$=$	$high^{auth}_{\{x\}}$
$I_{cls}(\texttt{[EAL6high]})$	$=$	$high^{biosig}_{\{x\}}$	$I_{cls}(\texttt{Bio2})$	$=$	$med^{auth}_{\{x\}}$
$I_{cls}(\texttt{[EAL7high]})$	$=$	$high^{biosig}_{\{x\}}$			

Table 5.3: Assignment of applications to access classes

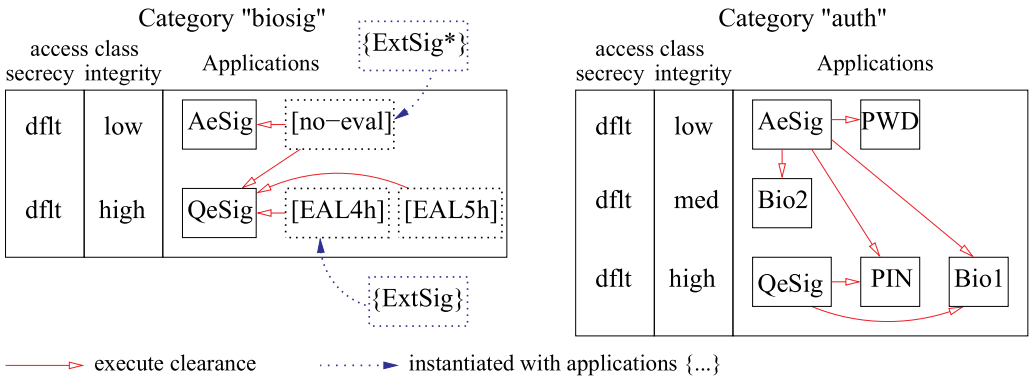


Figure 5.8: A selection of communication channels between applications in the `BioSig` example

{ExtSig}. From the BioSig point of view there is no difference between two differently evaluated applications as long as they provide proper credentials stating the sufficient evaluation level. Please note, that even higher evaluated applications are assigned to $high^{biosig}_{\{p\}}$. Hence, the application ExtSig is allowed to execute QeSig but not AeSig. The authentication procedure could be realized with certificates presented by the external application holding a public key as well as evaluation information as mentioned in the previous section.

Regarding the transport security of data between the dummy application and the external application we have to guaranty integrity of the transferred data, because the document or the hash value of the document to be signed shall not be altered or manipulated while transmitted. Confidentiality of the hash value is not required and thus, we are not concerned with secrecy access levels here and assign all applications to a default secrecy level dflt. A selection of possible communication channels and the classifications is given in Figure 5.8. There are more communication channels possible, for instance QeSig could execute an [EAL5h] instantiated application. Since QeSig is not designed to do so, we skip those channels. Please note, the application functionality does not intend those communications but they are basically allowed by the security policy. Those indirect communications channels can be omitted when assigning each application to its own access category and with the definition of explicit channel programs as it is shown in the next section.

So far, we worked out the first step in the execution chain, for instance ExtSig executes QeSig. The signature creation application QeSig has to authenticate the user in order to activate and generate the electronic signature. To keep the same security level it is only allowed to execute authentication applications of the same (or higher) integrity level. Again, in this scenario we are not concerned with secrecy, because the answer of an authentication application, which may be 'yes' or 'no', is not confidential but must be full of integrity. That is why QeSig is additionally assigned to the access class $high^{auth}_{\{x\}}$ and may choose PIN or Bio1. In this implementation we decided to create a second integrity access category for the authentication applications. This modularity allows us to use all these applications in other scenarios as well. One may think of a physical access control that wants to use one of the biometric authentications. Therefore we simply have to define one more integrity access category, e.g. $MultAC$ and do the assignments.

Next, we need to handle the external devices that are needed by all authentication applications, which also denotes the third execution step in the execution chain, e.g. ExtSig calls QeSig calls PIN calls ExtDevice. We again define new access categories for PIN pads *pinpad* and biometric capture de-

$S_{cls}(\texttt{PWD})$	=	$low^{pinpad}_{\{x\}}$	$I_{cls}(\texttt{PWD})$	=	$low^{pinpad}_{\{x\}}$
$S_{cls}([\texttt{no-eval}])$	=	$low^{pinpad}_{\{x\}}$	$I_{cls}([\texttt{no-eval}])$	=	$low^{pinpad}_{\{x\}}$
$S_{cls}(\texttt{PIN})$	=	$high^{pinpad}_{\{x\}}$	$I_{cls}(\texttt{PIN})$	=	$high^{pinpad}_{\{x\}}$
$S_{cls}([\texttt{EAL4h}])$	=	$high^{pinpad}_{\{x\}}$	$I_{cls}([\texttt{EAL4high}])$	=	$high^{pinpad}_{\{x\}}$
$S_{cls}([\texttt{EAL5h}])$	=	$high^{pinpad}_{\{x\}}$	$I_{cls}([\texttt{EAL5high}])$	=	$high^{pinpad}_{\{x\}}$
$S_{cls}([\texttt{EAL6h}])$	=	$high^{pinpad}_{\{x\}}$	$I_{cls}([\texttt{EAL6high}])$	=	$high^{pinpad}_{\{x\}}$
$S_{cls}([\texttt{EAL7h}])$	=	$high^{pinpad}_{\{x\}}$	$I_{cls}([\texttt{EAL7high}])$	=	$high^{pinpad}_{\{x\}}$
$S_{cls}(\texttt{Bio1})$	=	$high^{biodev}_{\{x\}}$	$I_{cls}(\texttt{Bio1})$	=	$high^{biodev}_{\{x\}}$
$S_{cls}([\texttt{EAL4h}])$	=	$high^{biodev}_{\{x\}}$	$I_{cls}([\texttt{EAL4h}])$	=	$high^{biodev}_{\{x\}}$
$S_{cls}([\texttt{EAL5h}])$	=	$high^{biodev}_{\{x\}}$	$I_{cls}([\texttt{EAL5h}])$	=	$high^{biodev}_{\{x\}}$
$S_{cls}([\texttt{EAL6h}])$	=	$high^{biodev}_{\{x\}}$	$I_{cls}([\texttt{EAL6h}])$	=	$high^{biodev}_{\{x\}}$
$S_{cls}([\texttt{EAL7h}])$	=	$high^{biodev}_{\{x\}}$	$I_{cls}([\texttt{EAL7h}])$	=	$high^{biodev}_{\{x\}}$
$S_{cls}(\texttt{Bio2})$	=	$high^{biodev}_{\{x\}}$	$I_{cls}(\texttt{Bio2})$	=	$med^{biodev}_{\{x\}}$
$S_{cls}([\texttt{EAL4m}])$	=	$high^{biodev}_{\{x\}}$	$I_{cls}([\texttt{EAL4m}])$	=	$med^{biodev}_{\{x\}}$
$S_{cls}([\texttt{no-eval}])$	=	$low^{biodev}_{\{x\}}$	$I_{cls}([\texttt{no-eval}])$	=	$low^{biodev}_{\{x\}}$

Table 5.4: Assignment of external devices to access classes

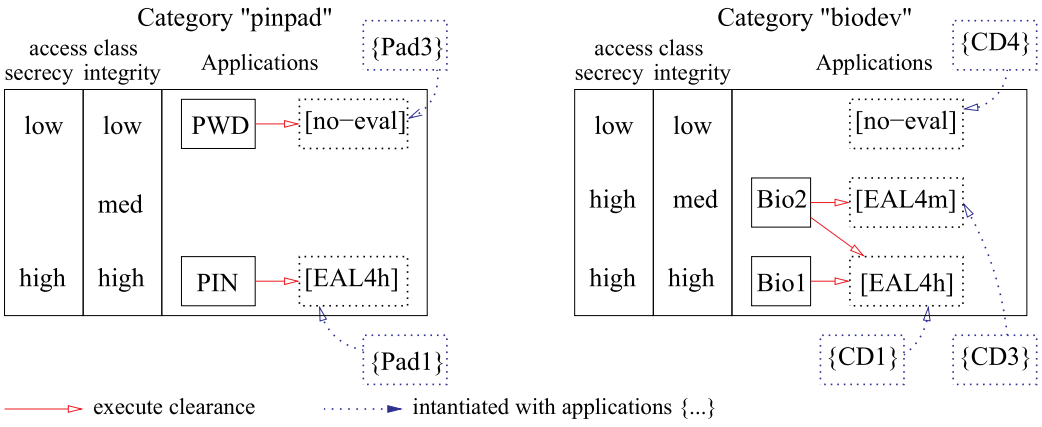


Figure 5.9: A selection of Communication channels between applications in the `BioSig` example (cont.)

vices *biodev*, which enlarges the set $IntC = \{biosig, auth, pinpad, biodev\}$. Because the PIN as well as the captured biometric data are very sensitive data and thus, must be handled confidentially, we define secrecy access categories and classes $SecC = \{pinpad, biodev\}$, $SecL = \{low_{\{x\}}, med_{\{x\}}, high_{\{x\}}\}$ and assign the access classes to the categories for integrity and secrecy

$$S_{ctg}(low_{\{x\}}) = S_{ctg}(high_{\{x\}}) = pinpad$$
$$S_{ctg}(low_{\{x\}}) = S_{ctg}(high_{\{x\}}) = biodev$$
$$I_{ctg}(low_{\{x\}}) = I_{ctg}(med)_{\{x\}} = I_{ctg}(high_{\{x\}}) = pinpad$$
$$I_{ctg}(low_{\{x\}}) = I_{ctg}(med_{\{x\}}) = I_{ctg}(high_{\{x\}}) = biodev\ .$$

Furthermore, all authentication applications need to be assigned to the access classes as shown in Table 5.4. We again assign all higher evaluated devices also to $high^{biodev}_{\{x\}}$ and $high^{pinpad}_{\{x\}}$, respectively.

Let us assume `PIN` executes a software implementation `Pad1` running on a host computer that uses the keyboard to get the user PIN, which is `EAL4high` evaluated (such applications really exist). Let us further assume a second `EAL4high` evaluated application `Pad2` that is a hardware implementation integrated in a smart card reader that provides a PIN pad. From the `PIN` application point of view there is no difference between the two external devices (or applications) as long as they provide proper credentials stating the sufficient evaluation level. It is $[\texttt{EAL4high}] = \{\texttt{Pad1}, \texttt{Pad2}\}$ in the pinpad category.

That means we treat external devices in the same way as external applications. If an external device, e.g. `Pad3`, can not present any evaluation it is assigned to the lowest level and thus can only be used by the `PWD` application. We give a few assignments and the resulting communication channels in Figure 5.9. Please note, the `PWD` application is also not permitted to execute the `Pad2` application because it is not cleared for secrecy access class `high`.

Furthermore, if an external device can not present a certificate at all and thus, can not be authenticated it is assigned to no access class and can only be used by as well not classified applications. One may think of external devices that are not equipped with security mechanisms and therefore are not able to establish a secure connection, e.g. to encrypt the transferred data. As a further example the biometric capture device `CD4` can not provide a sufficient evaluation and is therefore assigned to the secrecy and integrity access class *low*. Hence, the application `Bio2` has no permit to use `CD4`. On the other hand it is cleared to use the biometric capture devices `CD1` and `CD3`.

Lastly, we need to implement the functionality of authentication chaining, that is once the `QeSig` application is activated with `PIN` or `Bio1`, it can be activated by `Bio2` for ten times in succession. This is a requirement given in Section 2.3. Therefore, we again define a new integrity access category $IntC = \{authchain\}$, which again enlarges the already de-

ExtSig	⟶	QeSig	⟶	PIN	⟶	Pad1		
				...	⟶	Pad2		
		...	⟶	Bio1	⟶	CD1		
				...	⟶	CD2		
		...	⟶	PIN	⟶	Pad1		
		...	⟶	PIN	⟶	Bio2	⟶	CD1
ExtSig*	⟶	QeSig	⟶	...	see	above		
...	⟶	AeSig	⟶	PIN	⟶	Pad1		
				...	⟶	Pad2		
		...	⟶	PWD	⟶	Pad3		
		...	⟶	Bio1	⟶	CD1		
				...	⟶	CD2		
		...	⟶	Bio2	⟶	CD1		
				...	⟶	CD3		

Table 5.5: Some possible execution chains of the example BioSig

fined set. It is sufficient to assign one integrity access class to the category $I_{ctg}(low_{\{x\}}) = authchain$ as well as assigning the applications to the integrity access class $I_{cls}(\texttt{PIN}) = low_{\{x\}}^{authchain}$, $I_{cls}(\texttt{Bio1}) = low_{\{x\}}^{authchain}$ and $I_{cls}(\texttt{Bio2}) = low_{\{x\}}^{authchain}$. With this additional assignments Bio1 or PIN may execute Bio2, whereas QeSig is not allowed to execute Bio2. Of course, Bio1 and PIN need to be enabled to perform the chain.

A summary of possible execution chains is given in Table 5.5. Here, we assume ExtSig* with no evaluation, e.g. a PGP application, and hence $I_{cls}(\texttt{ExtSig*}) = low^{biosig}$. The execution chains represent the activation matrix in Table 2.1. As conclusion we can say that the implementation of the BioSig example in our model fits the requirements given by the informal description in Section 2.3.

5.5 Alternative design decisions

In the previous section we defined five access categories and assigned all applications to a single or to two categories and its access classes. The communication channels between the applications implicitly result from the rules given by the Bell/LaPadula and Biba models. A different approach in defining communication channels is to use *channel programs* as mentioned in Section 5.1.

Here, every single application is assigned to a single access category use-

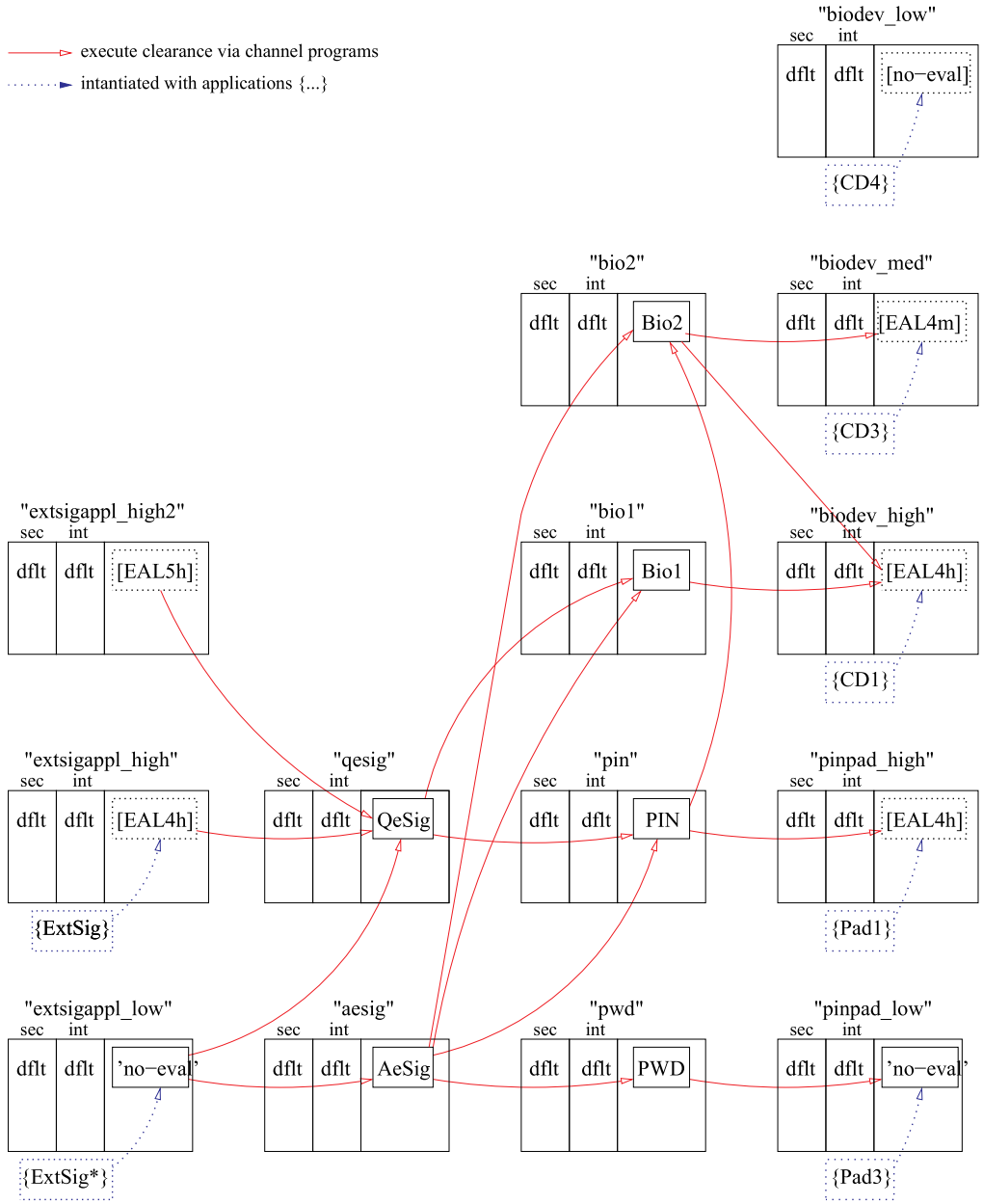


Figure 5.10: Alternative design of the `BioSig` example using channel programs only

fully named after the application, e.g. the `AeSig` application is assigned to access category *aesig*, `QeSig` to *qesig*, `PWD` to *pwd* and so forth. With this default classification there is no communication at all between the different applications. Desired communication channels can be established by explicitly defining *channel programs* that transfer information from one application to another. An alternative design of the `BioSig` example is given in Figure 5.10.

Within an access category an application may create subprograms and assign them to secrecy and integrity classes generated by the application. In our scenario we do not make use of it and assign all applications to a default access class `dflt`. Please note, a channel program can only transfer information from a single application or subprogram to another. There is no direct communication to other subprograms in the specific category unless a new channel program is defined. So, it makes it necessary to define all intended communication channels as shown in the figure.

We again create general access categories, e.g. `extsigappl_high`, to get external signature applications correctly assigned as long as they provide a sufficient evaluation level, e.g. `EAL4high` or higher. Because we want to keep the `ExtSig` application from executing the `AeSig` application, we only create one channel program between the [`EAL4high`] and the [`AeSig`] application and continue with all the other applications. The handling of external applications and the loading and assigning of new applications remain the same as in the previous design. It turns out that the alternative design in fact creates the same execution chain as shown in Table 5.5.

5.6 Conclusion

In the previous chapters we motivate the need for a new extended security policy for multi-applicative smart cards, which places the responsibility for the managing of external applications and external devices from the applications themselves on the operating system. We stress this by presenting a significant real-world case study `BioSig`.

The Extended Model of Security Policy introduced in this chapter takes the environment of a smart card into account. Therefore, we integrate external applications and external devices in the SMaCOS security model. The model is well-suited for this purpose because the underlying Bell/LaPadula and Biba models are well understood. A further promising approach could be the application of role-based access control policies.

Since in the SMaCOS model confidentiality and integrity requirements of the on-card applications are met by means of access control mechanisms

implemented by the operating system, this does not hold for external applications. Those communications are over an open network. The integration requires the authentication of the external applications and devices as well as the secure transmission of the data. Access rights of external applications and devices are then determined by secrecy and integrity levels, whereby integrity classes can be defined by evaluation levels of the Common Criteria or ITSEC.

It turns out that in both example implementations of the `BioSig` case study we take advantage of integrity access categories that have be seen of no practical relevance so far. The main difference between the two approaches of applying the `BioSig` case study is the implicit and explicit definition of the communication channels. One should be aware that the former design creates much more implicit communication channels, whereas the latter one creates exactly the explicitly defined ones. This might be an advantage and has to be paid with much more applications running on the card, because the channel programs are modeled as applications. The design decision must be done individually for every single application. The overall advantage of the extended security policy is the flexibility of designs. Please note, even a mixture of both approaches is possible and makes sense for individual applications.

Whether the specific design decision it is crucial for the security of the whole system that the assignments of all applications and channel programs is done properly. A false assignment may lead to unwanted information flow. This may also lead to new security tasks, e.g. the *cascade problem* of MLS systems. In other words, the extended security policy of the smart card operating design only provides the security functions that should be used properly.

On the other hand, the application provider as well as the smart card issuer heavily depend on the correctness of the security functions and they can not influence the functionality. To stress the reliability in the extended security policy we formally verify that the security functions implemented by the operating system indeed fulfill the security objectives. This has already been done for the SMaCOS security model by means of access control, which is illustrated in red full arrows in the figures of this chapter. It remains to verify the correctness of the new functionality realizing the external communication denoted in dotted blue arrows in the figures.

Before we perform the different formal verification tasks in the following chapters we give an overview of how formal methods can be applied in security analysis in the very next chapter.

Chapter 6

The analysis of security policies

We already identified potential *security threats* for multi-applicative smart cards in Section 3.2 on Page 19. Based on the threat analysis we defined *security objectives* in Section 3.3 that should be met by the smart card. Therefore, the smart card has to provide *security functions* implementing the security objectives and countering the security threats.

In this chapter we will analyze the relations between threats, objectives and functions in order to give arguments why and where formal methods can improve the reliability in a particular system design. Furthermore, we will have a brief look at how international security evaluation criteria reflect those relations.

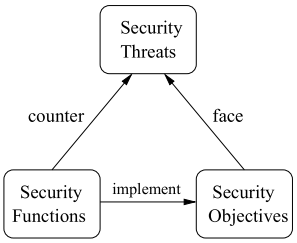
6.1 Security triangle relations

In Section 3.1 we introduce the *security policy*, which combines security objectives and security rules or functions. In general, a security policy defines objectives that can be either fulfilled by *organizational measures* or implemented by *technical security functions*. Please note, there are various meanings of the term security policy in the literature and a good discussion is given in (Ste91).

For example, a security policy of a company may demand that each employee has to revoke a lost smart card. Such scenarios can often be found when a smart card serves as an access control token without user authentication (in analogy to traditional keys). This objective can not be implemented by a function of the card (it can not revoke itself) but by an organizational rule. Another example regards passwords that should be known to the authorized user only. An organizational rule may state that passwords of employees have always to be kept secret and will never be asked by neither chiefs nor

system administrators. Those attacks are also known as *social engineering* and can not be countered by technical measures.

Since a smart card is a technical device it can only cover the technical part of a security policy, whereas organizational rules define the *security environment* for the proper use of the smart card or any other security product.



Therefore, we have carefully to distinguish a number of threats of different categories. We call the relations between security threats, objectives and rules or functions the *security triangle*. We already identified two different general security triangles, a *functional* and an *organizational triangle*. A simplified functional security triangle is shown beside.

But we can identify even more security triangles, which is determined by the level of abstraction. If we, for instance, treat the smart card as an entire system on a very abstract level, we may be satisfied with a single functional security triangle. In this case we are not concerned with where to place the security functions whether in the application level, the operating system level or the hardware level. But in the system engineering process a refinement might be necessary that additionally leads to refined or additional security triangles.

As an example, one may think of subjecting the smart card chip to a specific radiation in order to flip bits and to unauthorized change access rights. Moreover, one may simply *read out* confidential bits by dismantling the chip. Those kinds of physical threats of the smart card should be countered by hardware measures, which is known as *tamper-resistance* of processors. Furthermore, there are logical threats that can be countered by the operating system layer. We already gave logical threats summarized in Figure 3.2. The possible resulting security triangles are illustrated in Figure 6.1.

That is why threat analysis and the delimitation of security objectives is a very crucial part in security system engineering. Once we separated all security threats and objectives as well as functions and rules and therefore, built up the different security triangles, we are able to explicitly state which threats are countered by the security product and even more important which are not countered by it.

We are concerned with the security of a smart card operating system in a multi-applicative environment on a logic level and identified various security threats. As in all other smart card operating system designs we assume the underlying hardware to be trustworthy and hence, we do not take those threats into account. In opposition to the threat models of the mentioned smart card designs in Chapter 4, our threat model additionally considers the communication between on-card applications and the outside world. As

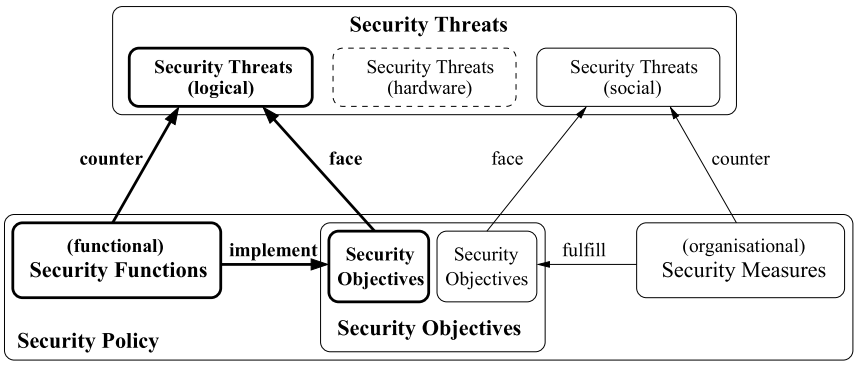


Figure 6.1: Possible security triangles

state of the art, the responsibility of those communications is placed on the applications of the smart card.

In the following we are mainly concerned with the triangle that is defined by the introduced extended model of security policy, which are the identified threats `ST1-ST11` (Section 3.2), the objectives `SO1-SO6` (Section 3.3) and functions `SF1-SF6` (Section 5.3). This security triangle regarding logical threats is marked in bold face in Figure 6.1.

Despite the individual design decision of a particular smart card system according to a specific threat model, there are always several security triangles that need to be analyzed. Three main questions have to be answered for every single security triangle as well as for the composed security triangle.

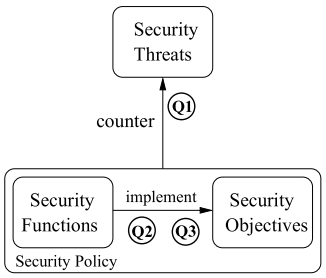
`Q1`: Does the security policy counter all identified security threats? (SP Effectiveness)

`Q2`: Do the security functions implement all security objectives? (SP Completeness)

`Q3`: Is the security policy sound? (SP Consistency)

We will work out the second question first that asks for the completeness of the security policy or in other words: Is there at least one security function that implements a security objective? There might be two or even more functions that implement a particular objective. This scenario leads us to the third question that asks for the consistency of all functions. We want cases to be excluded where functions may work against each other and hence miss their intention. So, `Q2` and `Q3` are related to the edge *implement* in the functional security triangle in Figure 6.1. If we are somehow convinced that the defined security policy is complete and consistent we have to ask if it

really counters all identified security threats, which is addressed by the first question. Thus, Q1 is related to the *counter* edge of the triangle.



For simplicity we will use a short notation to illustrate the security triangle, which combines the arrows *counter* and *face* to one single arrow as shown beside. The figure additionally shows the parts of the triangle that are addressed by the different questions. Once again, the questions need to be answered for all particular security triangles. The following sections and chapters are concerned with providing good arguments to answer the mentioned questions as precise as possible. We will motivate the use of formal methods in order to give unambiguous arguments for the quality of a developed security policy.

6.2 Application of formal methods

A first analytical look at the security triangle of the extended model of security policy is given in Table 5.2 on page 50, which shows the security functions implementing the security objectives and countering the security threats. It turns out that in the extended security model we do not provide any security function that counters security threats ST10 and ST11, which address covert channels and the transfer of illegal information via legal channels. These threats must be countered by other means. Furthermore, the matrix can be used to identify functions that may implement an objective but do not counter any threat. In this case we either forgot to identify an important threat or defined a wrong objective. This would have led to an additional row with no threat. So, the matrix helps us to strengthen the security triangle because it must always hold that a certain security function implements a security objective and counters a security threat.

Generally, the verbal description of the security triangle may suffice for a first impression of a chosen design of the product under consideration. Because natural language descriptions neither use a fixed formalism nor is the underlying semantics clearly stated they are insufficient to reflect technical requirements. Hence, the question arises whether a technical realization, e.g. the executable code, of an informally given security function also implements the corresponding security objective. The answer to this question can become arbitrary hard if we for instance consider the security function implemented in a programming language, where the function may built up of several subroutines implemented in a hundred lines of code. Even high level programming languages as C/C++ do not provide an unambiguous

semantic because the semantic is determined by a particular compiler. In consequence we would have to show that all particular realizations of a security function, e.g. in C++, indeed implement the intended function. This is of course not applicable in reality. Hence, we need to abstract from a specific implementation.

One of the key issues of formal methods is that they relay on fixed and exactly defined languages whose semantics are defined upon mathematical concepts. Generally, formal description techniques allow for precise abstractions that may serve as a *missing link* between informal requirement specifications and technical realizations. In conclusion, formal methods provide a mathematically grounded framework for

- the description of systems (the specifications),
- the formulation of postulated system properties and
- the verification of these properties.

The major advantage of applying formal methods is that proofs can be done with mathematical precision once and for all. Hence, it remains to show that a particular realization of a security function refines the abstract verified description. But how can formal methods help analyzing the security triangle? We work this question out in the next sections.

6.2.1 Formal models of security policies

In order to formally analyze the security policy we first have to formalize the security functions in a conceptual way that concentrates on the *what* the functionality must provide and not on *how* is implemented. A number of generic formal security models have been introduced that can roughly be categorized into *access control models* and *information flow models*. Nearly all are based on the concept of a state machine, which defines an initial state and rules for changing the state of the system called *state transitions*. A *state* is a representation of the system at one moment in time, which should capture exactly those aspects of the system relevant to the security policy. Hence, the verification task leads to the question if a particular (insecure) state is reachable from the initial state, which also known as verifying *safety properties* of the system.

The generic models differ in the modeling of the states and state transitions. Access control models control the access of *subjects* to *objects* via an access matrix. The most prominent models are the Bell/LaPadula model (BLP) (BL76), the Harrison-Ruzzo-Ullman model (HRU76), the Take-Grant

model (Sny81) and the Chinese Wall model (BN89) that deal with confidentiality requirements. Integrity requirements are addressed by the Biba model (Bib77), the Clark-Wilson model (CW87) and Role Based Access Control models first published in (FK92). A good introduction in those models is given in (Gol99).

In contrast to access control models, information flow models are generally concerned with the flow of information between objects. For example, in an ordinary access control model a subject s_i that is authorized to read object o_i may write the information of the object in a different object o_j with different access rights to o_i. This may violate the security policy because a subject s_j may get an access right to the information via object o_j that it is not intended to. Those unwanted information flows are analyzed via information flow models.

Denning (Den76) firstly proposed a definition of *information flow* that was later formalized using state machines, resulting in a number of *information flow predicates* as the *Noninterference* predicate by Goguen and Meseguer (GM84; GM82) or Rushby (Rus92). Further information flow predicates include *Restrictiveness* (McC87), *Nondeducibility* (Sut86) and *Separability* (McL94a). The predicates are defined in terms of *traces* of the systems, whereas a trace is a possible sequence of inputs and outputs that characterize the operations of the system.

Rushby (Rus92) additionally shows that security in the sense of BLP can be seen as a special case of noninterference because BLP defines a *Multi-Level Security Policy (MLS policy)*. Hence, information flow control can be achieved by access control mechanisms. In fact, the proposed *lattice model* by Denning (Den76) is an extension of the BLP model.

Generally, once the security functions are formalized as an abstract system specification, say Γ, in a specific *calculus* we can apply well defined deduction rules given by the calculus. This in turn makes the formalization of the security objectives as properties ϕ of the system specification Γ necessary. Logically, we have to provide a proof that the proposition ϕ can be *derived* from axioms Γ, typically denoted as $\Gamma \vdash \phi$. In terms of the state machine this means that all state transitions preserve the safety property. If we can show that all identified properties hold in the system specification we have an unambiguous argument of the completeness of the security triangle, which is addressed by question `Q2` Completeness of the previous section.

However, constructing those proofs can be arbitrary hard since $\Gamma \vdash \phi$ is in general not decidable. In Figure 6.2 the formalization and verification tasks of a security policy in terms of the security triangle are illustrated, whereas boxes with rounded corners denote informal descriptions and rectangular boxes denote formal descriptions.

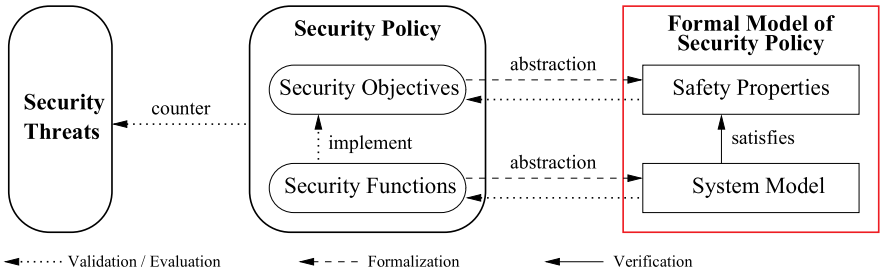


Figure 6.2: Formalization and verification of security policies

The question Q3 Consistency asks for the consistency of the security policy, which leads to the question of the consistency of the formal system specification and the properties. Logically, consistency means the absence of contradictions, i.e. there is no proposition ϕ such that $\Gamma \vdash \phi$ *and* $\Gamma \vdash \neg\phi$ hold at the same time. This means that all proofs based on Γ become meaningless because *all* formulas become derivable in case of inconsistency and thus, consistency is a property of the deductive apparatus.

However, for any computable axiomatic system that is powerful enough to formalize a certain amount of number theory the consistency of the axioms cannot be proved within the system itself (Göd31). To overcome the paradox we have to show that the axioms Γ' are either an extension by definition of calculus Γ or an interpretation within a calculus Γ, for which consistency has already been proven in literature. Mechanisms for consistent extension can be found in systems like Boyer&Moore (BM79), INKA (BHHW86), VSE (KUW95; HLsS$^+$96) or Isabelle (NPW02) and PVS (ORR$^+$96).

In conclusion, formal methods are suitable to show the completeness (question Q2) and consistency (question Q3) of the functional parts of the security policy in order to achieve a higher degree of confidence in the correctness of the security product. The question remains how formal methods can be used to analyze the effectiveness of the security policy, which is addressed by question Q1 Effectiveness.

In order to work out the question we have a brief look at the example of a computer pool for students at University. We assume the security policy of the administrator says that only authorized students are allowed to use the computer resources and printers because the administrator, however, identified several abuses of the computers in the past. In this scenario the *abuse of computers* denotes a security threat, and the rule *authorized students only* denotes a security objective. A generic security function could be the *authentication of every student*. In a very simple formal model of the security policy one could verify that whenever a student sends a print job to

the printer she successfully passed the authentication check. Here, the question of the effectiveness of the security policy is reduced to so called *safety properties*, which state that the system never enters an insecure state, e.g. a not successful authenticated student gains access to the printer. It might sound trivial in this example but in larger scenarios this very abstract level can be a good starting point for the development of a security product.

The SMaCOS security model has been verified in this fashion (SRS+00; SRSS99). Therefore, the authors define basic abstract data types for applications and files and the necessary assignment information of the secrecy and integrity level. Moreover, basic operating system commands like *read, write, create, load-appl, authenticate* on the data are defined together with a *system state* and *state transitions*, which are the symbolic execution of the system commands. Finally, it is proved that every system command respects the access rules, e.g. given by Bell/LaPadula und Biba, and that all ever loaded applications onto the card have been successfully passed the authentication check. The authentication itself is modeled as an abstract predicate that needs to be refined in a further development step. In other words, it has been shown that starting in a secure initial state there is no state reachable, where first an application can read or write data without the appropriate right and second has been loaded onto the card without a successful authentication. The verification of those safety properties is additionally denoted in red boxes in Figure 6.2.

On the other hand, the security policy and its analysis become more expressive if we, for instance, refine the authentication security function mentioned in the University example or in SMaCOS with a particular authentication method. One may think of a password based method or of the use of personalized smart cards in combination with a challenge-responds protocol. In such a refined model of security policy we have to take several attack scenarios into account in order to analyze the effectiveness. This has extensively been done in the context of the formal analysis of *security protocols*, which is discussed in the next section.

6.2.2 Formal analysis of cryptographic protocols

In the previous section we give a brief overview of how formal methods are applied to analyze security policies. A second field of formal analysis in the security context are security or cryptographic protocols. Generally, the goal of a security protocol is to provide various services between agents across an open network, e.g. authentication of agents or confidentiality of the transferred data. A good selection of those protocols can be found in (Sch96; MvOV97).

As in formal models of security policies we again have to formalize both the security functions in an abstract system specification $\Gamma^\star$ and the security objectives as properties $\phi^\star$ of the system $\Gamma^\star$. Additionally, we have to formalize the attacker. In the Dolev and Yao (DY81) scenario we assume the attacker to be able to read all data sent via the network, to create and send new data, to replay or delete data and to encrypt and decrypt data using keys taken from the knowledge base of the attacker. If we can formally derive a property, e.g. confidentiality of specific data, in such a model, we have an unambiguous argument of the effectiveness of the security functions. We refer to properties that should hold under a given threat model as *security properties*. Figure 6.3 shows the formalization and verification tasks in the attacker model scenario. The boxes with rounded corners again denote informal descriptions and rectangular boxes denote formal descriptions, whereby we stress the verification of security properties additionally in blue color.

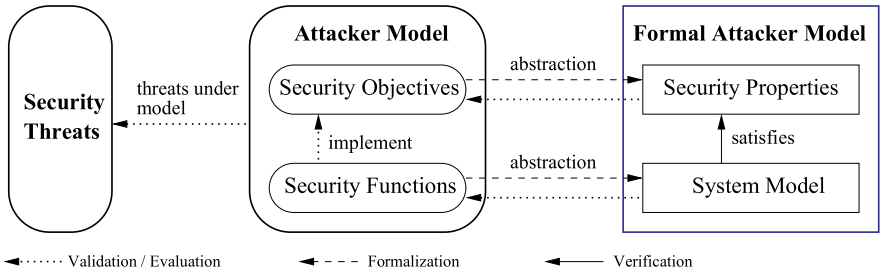


Figure 6.3: Formalization and verification of security properties

Generally, the expressiveness of the formal verification of a security model heavily depends on the capabilities of the attacker in a particular model. Much effort has been done in the field, which led to numerous formalisms. We can roughly identify three main approaches.

Believe logics reason about the state of believe of the agents involved during the execution of the protocol, whereas the attacker is not explicitly modeled. The BAN logic of authentication (BAN89) is the most prominent representative of this group and has been extended by the GNY logic (GNY90). *State enumeration* techniques systematically enumerate possible runs of a given protocol including the attacker events. In (RSG$^+$00) the use of the process algebra CSP and the model-checker FDR is proposed as well as a good overview of formal analysis of security protocols. Another long known model-checker is the NRL Protocol Analyzer by Meadows (Mea96). A further approach is performing *inductive proofs* about protocol traces modeled as a sequence of events as firstly proposed by Paulson (Pau98). Bella extended Paulson's approach to allow, among others, the modeling of specialized smart

card protocols (Bel00; Bel07).

All the formal approaches assume the used *cryptographic primitives* to be *secure*. Primitives are the security functions as symmetric and asymmetric algorithms, hash functions, random number and key generators or digital signature algorithms. In this sense secure means that, e.g. decrypting a cipher text is possible only with the corresponding key and an attacker has at last to perform a brute force attack on the key space. Furthermore random number generators are assumed to provide an unpredictable sequence of numbers.

This abstraction from the underlying cryptographic primitives should focus on the protocol properties itself and should make it independent of specific functions. On the other hand this abstraction is the main criticism because it does not allow to include a probabilistic attacker. The Backes-Pfitzmann-Waidner Model tries to bridge this gap (BPW03; SBB$^+$06). Despite the criticism formal analysis of security protocols proved very helpful in detecting flaws in their design.

6.2.3 Safety versus security properties?

In this section we will demonstrate how formal methods can be applied to the security triangle defined by the Extended Model of Security Policy in order to strengthen the reliability in it. In the last sections we figured out that the security functions have to be formalized as a system specification as well as the security objectives have to be formalized as properties of the specification.

It turns out that security functions `SF1` to `SF4`, which are the access control mechanisms by Bell/LaPadula and Biba, are under the full control of the operating system. They shall implement the security objectives `SO2` (application isolation), `SO3` (application control) and `SO4` (communication within the card). For instance, in this context application isolation (`SO2`) means that data of application A are completely confidential for application B. Furthermore, if communication is allowed between A and B the integrity of the transferred data is required.

In our scenario of a smart card operating system the security objectives, as data confidentiality or data integrity, can be fulfilled by access control mechanisms, e.g. by a reference monitor. Because the operating system fully controls *all* access, we do not need to take an attacker model into account. Please note, we assume the underlying hardware to be tamper resistant. If we model the system as a state transition system the we have to show that there is no behavior of the operating system that violates the access conditions. In other words, every access of a particular application to files on the smart card is controlled by the operating system and checked for authorization. Hence,

the objectives become *safety properties* in the formal model.

As already mentioned in section 6.2.1 the verification has been done for SMaCOS in (SRS+00). Furthermore, the authors verified the authenticity check for every application loaded on the card, which means every application on the card successfully passed the authenticity check. This is expressed in `SO1` (download of applications). The authentication itself is left open in the verified SMaCOS model and may require the integration of a security protocol in a further refinement step.

The security objectives `SO5` (external application communication) and `SO6` (external device communication) state that communication between an on-card application and an external application or device is allowed only if both partners agreed on it. This communication is via an open network and therefore not under the full control of the smart card operating system. Security function `SF6` requires entity authentication as well as data confidentiality and data integrity for the data transferred via the network. Because the operating system is just part of the open network we cannot achieve the objectives via access control mechanisms. The integration of security protocols is necessary. Therefore, in contrast to the above mentioned objectives we are confronted with the verification of a security protocol. Hence, objectives `SO5` and `SO6` become *security properties* in a formal model.

Figure 6.4 illustrates the possible formalization and verification tasks in the entire functional security triangle. On the left hand side we identify all security threats that should be countered by the security policy that itself divides in security objectives and security functions. We already introduced them as security triangles. Please note, there are always undiscovered threats. The dotted arrows signal that the correctness of the relation under consideration can be *evaluated* only or in other words, formal methods can not be applied. For instance, the correctness and consistency of informally given security objectives and functions can not be formally verified, obviously. In the figure an informal description is denoted by a box with rounded corners and formal descriptions are denoted in rectangular boxes.

In consequence, we can not formally verify that the informally given security policy counters the informally given security threats as depicted in the figure. But there is a little way out as we will see later on. In order to apply formal methods we need to *formalize* the objectives and functions, which is denoted as dashed arrows. In the formalization process we additionally abstract from the informal description. It is again an evaluation task to check if the abstraction does neither hide important functions nor adds new functions. Once we formalized the security policy we can start verifying, which is to reason in the particular formal calculus, and provide a correctness proof. This is denoted by a full arrow. The figure shows two of those formalizations,

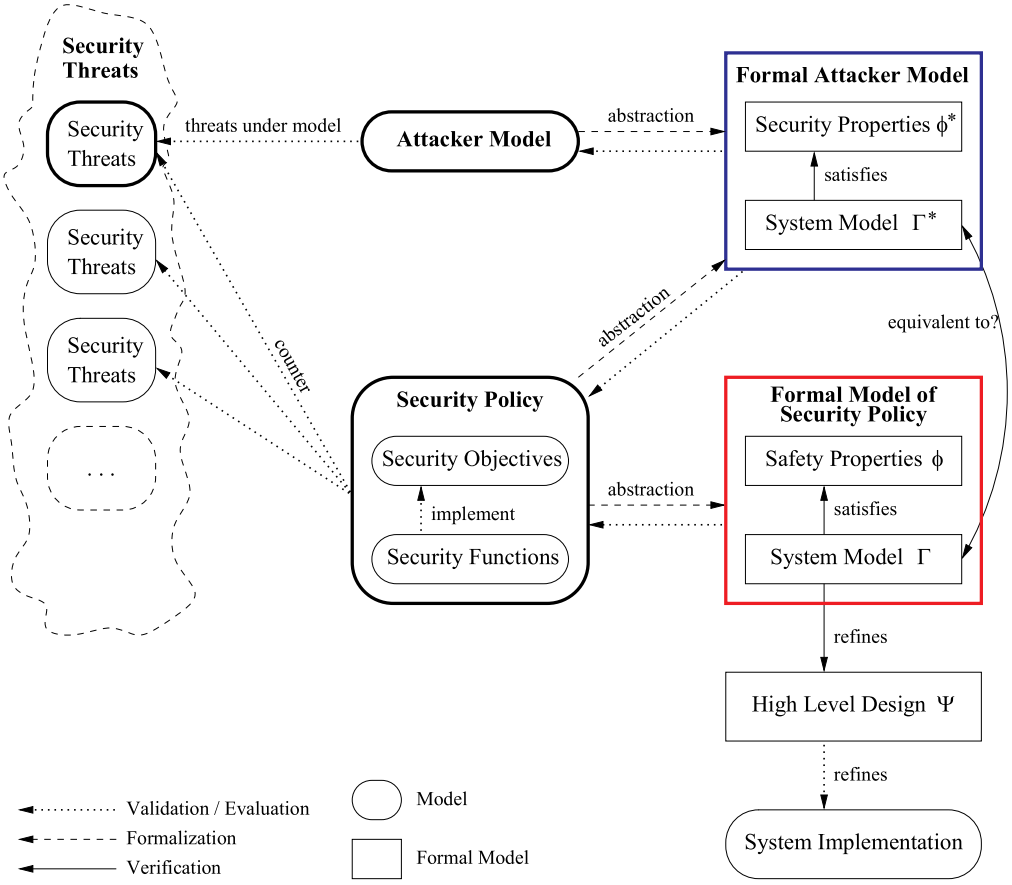


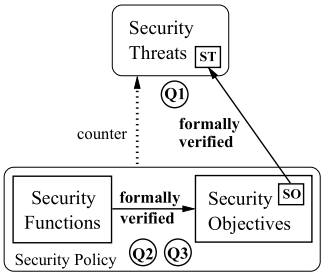
Figure 6.4: Possible formalization and verification tasks

one is the formal model of security policy and the other is the formal attacker model that we describe later on.

In a further step of the development process the verified abstract formal system model Γ has to be refined by a high level design, say Ψ, of the system. This refinement is logically and in the 'safety' sense an implication $\Gamma \rightarrow \Psi$. We say that all behaviors of Γ are also behaviors of Ψ because Γ is more general than Ψ. There might be even more refinement steps, at least as long as the formal apparatus carries. In the ideal case we end at source code level, but in this work we concentrate on the design level.

Figure 6.4 also shows the gap between security protocol analysis and security policy analysis because as state of the art we cannot deduce security and safety properties in the same formalism. Hence, we again have to abstract and formalize from the security policy, say a system specification $\Gamma^{\star}$, in order to perform the security property verification. But it is worth the effort, because with a formal attacker model in the context of security protocol

analysis we can formally verify that at least a subset of the security policy counters a well defined subset of the security threats. In Figure 6.4 the formal verification tasks regarding the extended security policy are illustrated, whereby the verification of safety properties is denoted in red color and the verification of security properties is denoted in blue color.



This is additionally displayed in the figure beside that shows the possible formal verifications in a simplified way. The subset of threats ST is determined by the chosen attacker model and thus determines the subset of security objectives SO. The challenge in this general scenario is to provide a formal proof of equivalence between both system specifications Γ and $\Gamma^\star$, so that the specification Ψ also inherits the properties $\phi^\star$ of specification $\Gamma^\star$. In this work we provide a general methodology to bridge the gap.

Finally, we have to admit that in general finding security threats and defining security objectives is a rather creative (informal) procedure that takes a lot of experience in the field of security engineering. Obviously, the security policy cannot counter threats that have not been identified. No security engineer can develop a security product that is secure against *all* attacks because there might by successful attacks that have neither been published nor discovered by himself. Hence, the security provided by a security product is always relative to the state of the art.

6.3 Security evaluation criteria

Security evaluation criteria have long been used in security engineering and started in 1985 with the publication of the *Trusted Computer Security Evaluation Criteria (TCSEC)* (DoD85) by the Department of Defense of the U.S. called the *Orange Book* (because of the orange color of the book cover). It was primarily used for the evaluation of products developed for government use. Additional criteria are published in FIPS 140-1 (FIP01) by the National Institute of Standards and Technology (NIST) for cryptographic modules, which include both hardware and software components.

Other countries developed their own criteria as the *Canadian Trusted Products Evaluation Criteria (CTPEC)* (CSI85) in Canada. In the UK this includes CESG Memorandum Number 3 (CES89) developed for government use as well as proposals of the Department of Trade and Industry for commercial IT security products called the "Green Book" (DTI89). In Germany the German Information Security Agency published a first version of its own

criteria in 1989 (BSI89). At the same time criteria were being developed in France, the so-called "Blue-White-Red Book" (Ser89).

In 1991 France, Germany, the Netherlands and the United Kingdom agreed on harmonized criteria called *European Information Technology Security Evaluation Criteria (ITSEC)*(ITS91). A further harmonization of the TCSEC, CTPEC and ITSEC in 1995 led to the international *Common Criteria for Information Technology Security Evaluation (CC)*, which became the most used criteria in commercial IT security products.

We have a look at the Common Criteria in the version 2.1 of August 1999 (CC99) in order to investigate how the analysis of the security triangle is reflected by it. An excellent comparison of the ITSEC and CC regarding the use of formal methods is given in (BSI04b). Generally, the Common Criteria pursue three main aims:

- Increasing the quality of security products;
- Making different security products of similar kind comparable;
- Certifying the quality of the product by an independent third party under approved security criteria.

We are mainly concerned with the first item and the analysis of the security triangle. The Common Criteria pay attention on the security triangle even the term itself is not used. They require both the postulation of a security policy and arguments for the effectiveness, correctness and consistency of the security product under evaluation, the *Target Of Evaluation (TOE)*.

Therefore, the Common Criteria provide a kind of *building set* of predefined security threats, objectives and functions. Generally, each security product defines its own security policy because it mostly addresses different security threats under different security environment assumptions. In the CC this is expressed in the *Security Target (ST)*. A security product is always evaluated against its ST. Hence, what is secure under the policy of a certain product may be insecure under a different policy of another product.

On the other hand, we can identify groups of products that share the same intention. To reflect this the CC additionally define general *Protection Profiles (PP's)* that address a common group of security products as smart cards (PP01). The instantiation of a generic PP is the Security Target (ST) belonging to a specific security product.

We already mentioned that the CC provide a kind of building set that is divided into two sets. Part 2 of the CC, *Security Functional Requirements*, establishes a set of *functional components* as a standard way of expressing the Functional Requirements for TOEs. The components are categorized in sets of *functional components*, *families*, and *classes*.

Part 3 of the CC, *Security Assurance Requirements*, establishes a set of *assurance components* as a standard way of expressing the Assurance Requirements for TOEs that are also categorized in sets of *assurance components*, *families* and *classes*. Part 3 additionally presents *Evaluation Assurance Levels (EAL's)* that define the predefined CC scale for rating assurance of TOEs. The CC define seven assurance levels from the lowest `EAL1` to the highest `EAL7` level, whereas the confidence in the correctness of the IT product increases with the levels.

In a very simplified way one can say that the set of functional requirements can be used to describe *what* functionalities the TOE should provide, whereas the set of assurance requirements can be used to describe *how* the TOE provides the functionalities. Hence, the latter determines the quality of the engineering process and thus the quality of the product. The main definitions regarding the security triangle are given in Part 2 by the *Functional Requirements Paradigm* that we cite in the following. The exact source of citation is given beside the text in terms of the part and paragraph.

> TOE evaluation is concerned primarily with ensuring that a defined *TOE Security Policy (TSP)* is enforced over the TOE resources. The TSP defines the rules by which the TOE governs access to its resources, and thus all information and services controlled by the TOE. — Part 2, 14

> The TSP is, in turn, made up of multiple *Security Function Policies (SFPs)*. Each SFP has a scope of control, that defines the subjects, objects, and operations controlled under the SFP. The SFP is implemented by a *Security Function (SF)*, whose mechanisms enforce the policy and provide necessary capabilities. — Part 2, 15

> Those portions of a TOE that must be relied on for the correct enforcement of the TSP are collectively referred to as the *TOE Security Functions (TSF)*. The TSF consists of all hardware, software, and firmware of a TOE that is either directly or indirectly relied upon for security enforcement. — Part 2, 16

The CC also define the term security policy and divide it into subpolicies. The threat analysis and the postulation of security objectives is required to be described in the Protection Profile and Security Target, which includes a detailed description of the security environment. The analysis requirements of the PP and security policy are given in Part 3 *Security Assurance Requirements*, whereas three assurance classes are of special interest:

- class `APE` Protection Profile Evaluation,
- class `ASE` Security Target Evaluation and
- class `ADV` Development.

In the following we will have a look at the requirements given by these assurance classes. Because Security Targets are an instantiation of a generic Protection Profile we only concentrate on the class APE Protection Profiles. The syntax of the CC requirements is

<class>_<family>.<component>.<paragraph>{D,C,E},

whereas D denotes *actions* to be taken by the developer, C denotes requirements on the *content* of the documents generated by the actions and C denotes *actions* to be taken by the evaluator.

The family APE_OBJ of the class APE gives requirements for security objectives.

Developer action elements:

APE_OBJ.1.1D The PP developer shall provide a statement of security objectives as part of the PP.

APE_OBJ.1.2D The PP developer shall provide the security objectives rationale.

Content and presentation of evidence elements:

APE_OBJ.1.1C The statement of security objectives shall define the security objectives for the TOE and its environment.

APE_OBJ.1.2C The security objectives for the TOE shall be clearly stated and traced back to aspects of the identified threats to be countered by the TOE and/or organizational security policies to be met by the TOE.

APE_OBJ.1.3C The security objectives for the environment shall be clearly stated and traced back to aspects of identified threats not completely countered by the TOE and/or organizational security policies or assumptions not completely met by the TOE.

APE_OBJ.1.4C The security objectives rationale shall demonstrate that the stated security objectives are suitable to counter the identified threats to security.

APE_OBJ.1.5C The security objectives rationale shall demonstrate that the stated security objectives are suitable to cover all of the identified organizational security policies and assumptions.

The definition of the security environment in the PP/ST of the IT product under consideration is the threat analysis and leads to identified threats. These threats are to be countered by organizational and technical measures, which is also defined in the PP/ST. Hence, the PP/ST exactly defines the security objectives to be met by the IT product and to be met by other means. In the security triangle this is denoted via different subtriangles in the arrows *faces* in Figure 6.1. The analysis of these relations and the answer to question Q1 Effectiveness is put in a separate evaluation of the

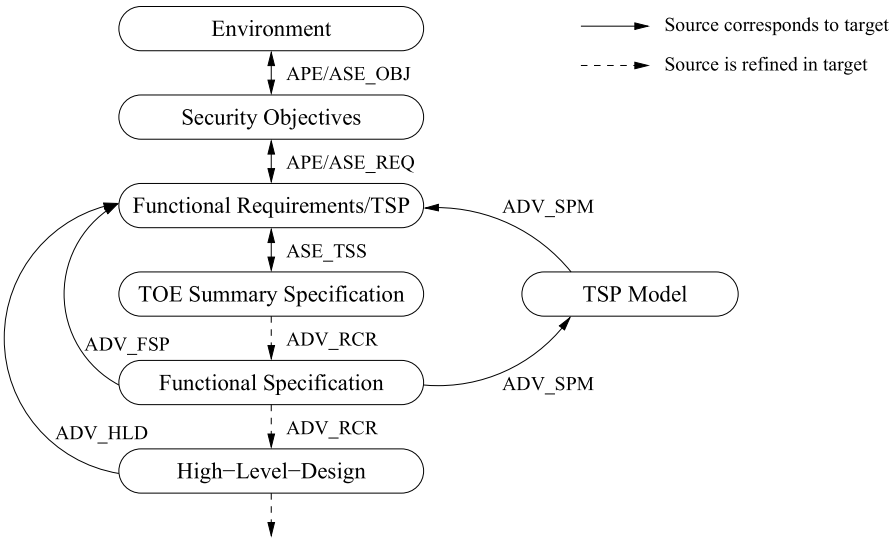


Figure 6.5: Relationships between TOE representations and requirements

particular Protection Profile. We already mentioned in Section 6.2.3 that finding security threats and the definition of security objectives is a rather creative (informal) procedure. The careful evaluation of a PP should increase the confidence in the effectiveness of the defined security objectives. The CC do not require the use of formal methods to analyze these relations as it is possible in the analysis of cryptographic protocols.

The security objectives, in turn, are refined into a set of *Security Requirements* for the TOE, which will ensure that the TOE can meet its security objectives. The family `APE_REQ` requires a statement that the Security Requirements are suitable to meet the security objectives and that they are complete, coherent and internally consistent. The statement shall be provided in informal style. In Figure 6.5 the assignments of the families to the idealized development process are illustrated as given in the CC.

The TOE *Summary Specification* provided in the ST defines the instantiation of the security requirements for the TOE. It provides a high-level definition of the security functions claimed to meet the functional requirements, and assurance measures taken to meet the assurance requirements. In this sense the security requirements together with the summary specification implicitly define the Security Policy of the TOE in the ST. The family `ASE_TSS` again requires a rationale that demonstrates the suitability of the IT security functions to meet the TOE Security Functional Requirements. The functions shall be defined in an informal style to a level of detail necessary for understanding their intent.

The *Functional Specification* is again the refinement of the Summary Specification. The assurance class ADV defines requirements for the stepwise refinement of the TSF from the TOE Summary Specification given in the ST down to the actual implementation. Therefore, the class ADV defines four development levels in separate families: ADV_FSP Functional Specification, ADV_HLD High Level Design, ADV_LLD Low Level Design and HDV_IMP Implementation Representation. Each of the resulting TSF representations provide information to help the evaluator determine whether the functional requirements of the TOE have been met. The family ADV_RCR states requirements for the correspondence proofs of the different representations. It distinguishes an informal, semiformal and formal style of the correspondence proof in components ADV_RCR.1/2/3, respectively. Additionally, Figure 6.5 shows a family ADV_SPM Security Policy Modeling.

The analysis of the relation between Functional Requirements and Functional Specification is required in the families ADV_FSP and ADV_SPM and addresses the question Q2 Completeness and question Q3 Consistency of the Security Policy. The required efforts in analyzing these relations is expressed in the assurance levels EAL1-7 and increases with the levels.

The explicit statement of a security policy model is required in assurance level EAL4 and higher. In the lower levels such a statement of the TOEs Security Policy is not required:

Part 3, 302

> The TOE security policy (TSP) is the set of rules that regulate how resources are managed, protected and distributed within a TOE, expressed by the TOE security functional requirements. The developer is not explicitly required to provide a TSP, as the TSP is expressed by the TOE security functional requirements, through a combination of security function policies (SFPs) and the other individual requirement elements.

In Table 6.1 the required components and families of class development ADV of every EAL is given. Since the explicit statement of a Security Policy TSP is firstly required in EAL4, in EAL1-3 the TSP is required to be implicitly expressed in the family Functional Requirements ADV_FSP. The family defines four components that describe the requirements on the Functional Specification. Roughly, component ADV_FSP.1 and component ADV_FSP.2 require an *informal* description of the *main* and *all*, respectively, functionalities, whereas component ADV_FSP.3 and component ADV_FSP.4 require a *semiformal* and *formal* description of *all* functionalities, respectively.

> *Developer action elements:*

ADV_FSP.1.1D

> The developer shall provide a functional specification.

> *Content and presentation of evidence elements:*

Assurance Family	Assurance Components by EAL						
	EAL1	EAL2	EAL3	EAL4	EAL5	EAL6	EAL7
ADV_FSP	1	1	1	2	3	3	4
ADV_HLD	-	1	2	2	3	4	5
ADV_IMP	-	-	-	1	2	3	3
ADV_INT	-	-	-	-	1	2	3
ADV_LLD	-	-	-	1	1	2	2
ADV_RCR	1	1	1	1	2	2	3
ADV_SPM	-	-	-	1	3	3	3

Table 6.1: Components, families and classes of the EAL's

The functional specification shall describe the TSF and its external interfaces using an informal style. ADV_FSP.1.1C

The functional specification shall be internally consistent. ADV_FSP.1.2C

The functional specification shall describe the purpose and method of use of all external TSF interfaces, providing details of effects, exceptions and error messages, as appropriate. ADV_FSP.1.3C

The functional specification shall completely represent the TSF. ADV_FSP.1.4C

Evaluator action elements:

The evaluator shall determine that the functional specification is an accurate and complete instantiation of the TOE security functional requirements. ADV_FSP.1.2E

In the following the additional requirements of components ADV_FSP.2/3/4 are denoted in bold face. We only give the paragraphs of the higher component that describe changes to the previous (lower) component. All not mentioned paragraphs of a previous component remain the same in the higher component.

The functional specification shall describe the purpose and method of use of all external TSF interfaces, providing **complete** details of **all** effects, exceptions and error messages. ADV_FSP.2.3C

The functional specification shall include rationale that the TSF is completely represented. ADV_FSP.2.5C

The functional specification shall describe the TSF and its external interfaces using a **semiformal style, supported by informal, explanatory text where appropriate**. ADV_FSP.3.1C

ADV_FSP.4.1C The functional specification shall describe the TSF and its external interfaces using a **formal** style, supported by informal, explanatory text where appropriate.

As shown in Table 6.1 the Functional Specification must be provided in an informal style in assurance levels EAL1-4, whereas in EAL1-3 only the main functionalities have to be considered. In EAL5-6 this must be provided in a semiformal style and in EAL7 in an formal style for all functionalities. A formal analysis of the Security Policy and hence, a formal proof of the completness and soundness as addressed by questions Q2 Completeness and Q3 Consistency is not required by the family FSP. The evidence that the Functional Specification accurately and completely instantiates the Security Requirements has to be provided in an informal style via a so called rationale.

The formal treatment of the Security Policy is required in family ADV_SPM for a subset of the Functional Specification that is determined by the state of the art. It is the objective of this family to provide additional assurance that the Security Functions in the Functional Specification enforce the policies in the TSP. This is accomplished via the development of a *Security Policy Model (SPM).*

Part 3, 367 While a TSP may include any policies, TSP models have traditionally represented only subsets of those policies, because modeling certain policies is currently beyond the state of the art. The current state of the art determines the policies that can be modeled, and the PP/ST author should identify specific functions and associated policies that can, and thus are required to be, modeled. At the very least, access control and information flow control policies are required to be modeled (if they are part of the TSP) since they are within the state of the art.

Part 3, 368 Foreach of the components within this family, there is a requirement to describe the rules and characteristics of applicable policies of the TSP in the TSP model and to ensure that the TSP model satisfies the corresponding policies of the TSP. The "rules" and "characteristics" of a TSP model are intended to allow flexibility in the type of model that may be developed (e.g. state transition, non-interference). For example, rules may be represented as "properties" (e.g. simple security property) and characteristics may be represented as definitions such as "initial state", "secure state", "subjects" and "objects".

Such a formal Security Policy Model is required in assurance levels EAL5-7 in addition to the requirements given in the family ADV_FSP. Despite a semiformal SPM is defined in ADV_SPM.2 it is not required in the assurance levels, whereas an informal SPM is demanded in EAL4.

Developer action elements:

The developer shall provide a TSP model.	ADV_SPM.1.1D
The developer shall demonstrate correspondence between the functional specification and the TSP model.	ADV_SPM.1.2D

Content and presentation of evidence elements:

The TSP model shall be informal.	ADV_SPM.1.1C
The TSP model shall describe the rules and characteristics of all policies of the TSP that can be modeled.	ADV_SPM.1.2C
The TSP model shall include a rationale that demonstrates that it is consistent and complete with respect to all policies of the TSP that can be modeled.	ADV_SPM.1.3C
The demonstration of correspondence between the TSP model and the functional specification shall show that all of the security functions in the functional specification are consistent and complete with respect to the TSP model.	ADV_SPM.1.4C

We again give the additional requirements of the components ADV_SPM.2/3 in bold face. All not mentioned components remain the same.

The TSP model shall be **semiformal**.	ADV_SPM.2.1C
Where the functional specification is at least semiformal, the demonstration of correspondence between the TSP model and the functional specification shall be semiformal.	ADV_SPM.2.5C
The developer shall demonstrate **or prove, as appropriate**, correspondence between the functional specification and the TSP model.	ADV_SPM.3.2D
The TSP model shall be **formal**.	ADV_SPM.3.1C
Where the functional specification is formal, the proof of correspondence between the TSP model and the functional specification shall be formal.	ADV_SPM.3.6C

To summarize, we can identify the security triangle in the Common Criteria methodology as well as requirements to analyze the triangle relations. In Figure 6.5 the idealized development methodology of an IT product of the CC is illustrated, which generally follows the waterfall approach (even the CC do not assume any specific methodology). The approach propagates the stepwise refinement from a very abstract system specification of the product towards executable code. The starting point in the CC methodology is the TOE Summary Specification given in the Security Target that shall meet the defined Functional Requirements that represent the Security Objectives.

In this way the CC define the Security Policy and is strongly concerned with the analysis of it. Therefore, in the lower assurance levels EAL1-4 the Completeness (question Q2) and Consistency (question Q3) of the Security Policy must be shown via informal rationales.

A formal analysis is firstly required in EAL5 with the formal Security Policy Model that should cover a subset of the Security Policy. A comprehensive formal treatment of the Functional Specification and of the High-Level-Design as well as the correspondence proofs between both representations is additionally required in assurance level EAL7, whereas this formal treatment primarily addresses the correctness of the refinement steps. Hence, we can say the CC indeed reflect the formal analysis of the Security Policy and strongly addresses questions Q2 Completeness and Q3 Consistency.

As worked out in Section 6.2.3 Security Policy Models are generally concerned with the verification of safety-properties. The formal analysis of security protocols and hence, the verification of security properties is not explicitly addressed by the CC. They indirectly propagate the use of security protocols, e.g. in families FCO_NRR Non-Repudiation of Receipt, FIA_UAU User Authentication, FTP_ITC TSF Trusted Channel and FPT_SSP State Synchrony Protocol, but do not require the analysis of the protocol. Especially in distributed systems those protocols are crucial for the quality of the entire system.

Generally, the analysis of the Effectiveness (question Q1) of the Security Policy is put in the informal evaluation of the Protection Profile and Security Target. Therefore, the CC propagate the use of an additional intermediate level in the development process, the TOE Summary Specification. Furthermore, it does not become clear why correspondence proofs between the more concrete design levels (HLD, LLD, IMP) and the Functional Requirements are necessary. With a powerful formally verified Security Policy Model it would suffice to formally proof that a lower specification level correctly refines the more abstract specification level. This is partly considered in the 3rd version of the Common Criteria.

6.4 Conclusion

This chapter aims at the analysis of the relations between security threats, security objectives and security functions that are all together the basis for the development of dependable secure systems. Therefore, we look at the three of them in a triangle relation and call this the Security Triangle. Although those relations are often recognized individually the term is rarely used but allows a more comprehensive view on the topic. The analysis of

the security triangle is central for the quality of a security product and leads to three questions: Q1 Effectiveness, Q2 Completeness and Q3 Consistency of the security policy.

As more precise these questions can be answered as higher the quality of the security product can get. Formal methods can help to give unambiguous answers to the questions, whereas two main fields of applying formal methods can be identified: formal models of security policy and formal analysis of cryptographic protocols. While the former methods analyze the security policy and address questions Q2 Completeness and Q3 Consistency, the latter are concerned with attacker models and address question Q1 Effectiveness. To distinguish properties to be verified in the different fields we name the former safety properties and the latter security properties.

The formal analysis of the Extended Model of Security Policy requires the verification of safety and security properties. Because the corresponding formal models are generally differently defined there is a lack in combining them in a formal way. Research has to be done to bridge this gap, which we will discuss in the next chapters.

The international security evaluation criteria Common Criteria pay attention on the Security Triangle. In the higher assurance levels they require the formal analysis of the Security Policy, but the verification of security properties in the context of the analysis of cryptographic protocols is not considered by the Common Criteria.

In the following chapters we will first formalize a Simple Smart Card Model that manages the external communication in order to verify safety properties. Then we verify security properties of a certain cryptographic protocol in order to eventually combine both models in one single model.

Chapter 7

Safety properties of state-based system specifications

When specifying a system using formal methods we can identify three main approaches. Property-oriented approaches like algebraic specifications concentrate on a functional system point of view, whereas model-oriented approaches are mainly concerned with states and state transitions and hence, focus on the intended behavior of the system. To take concurrency aspects into account the state-based methods have been enhanced with temporal expressions. The most prominent representatives of algebraic methods are Larch and LOTOS. The system Z is often used for model-oriented methods, whereas LTL, CTL and TLA are temporal formalisms.

Powerful tools have been developed to support modeling and reasoning in the mentioned formalisms, which is crucial in modern software engineering and in industrial sized development projects. Since we use the interactive theorem prover VSE-II (KUW95; HLsS$^+$96) that provides a temporal logic based on TLA, we first give an introduction into TLA in order to concurrently specify a model of a simple smart card operating system.

7.1 The Temporal Logic of Actions (TLA)

Lamport's TLA is a logical formalism providing means for the description of reactive systems in terms of state and state transitions as well as providing proof rules for reasoning about the system. Typical proof goals include the verification of system properties, mainly safety and liveness properties, and refinement proofs stating that one system (an implementation) refines another system (an abstract specification). The system specification and its properties together with a refined system specification and its refinement

proofs are all presented within the logic. Many papers have been published about TLA and most literature can be obtained from the home page of L. Lamport.[1] A comprehensive introduction into TLA from a logical perspective and from user perspective can be found in (AM96) and (Mer98), respectively. In the following we give an introduction into TLA in order to provide the basics to understand the system specification of a Simple Smart Card Model given in Section 7.2.

7.1.1 The basic formalism

In TLA we distinguish three kinds of formulas: *state formulas* describe *system states*, *action formulas* describe state transitions and *temporal formulas* describe system *behaviors* that are sequences of states. The syntactical structure of the formulas is that temporal formulas are used to build action formulas and they are used to build state formulas. The basic layer of the four level hierarchy is given by state independent formulas called *constants*.

In the following we assume a language $\mathcal{L}_P$ of classical first-order logic that defines function and predicate symbols together with an infinite set Var of variables. Let the set Var to be partitioned into the infinite and disjoint subsets Var_r and Var_f of *rigid* and *flexible* variables and let additionally be $Var'_f = \{v' : v \in Var_f\}$ the set of *primed flexible* variables. We call $Var_e = Var \cup Var'_f$ the (extended) set of rigid, flexible and primed flexible variables. A *transition function* is a first-order expression built from the predicate and function symbols of $\mathcal{L}_P$ and variables from Var_e. A *transition predicate*, which we will call an *action* later on, is a boolean-valued transition function. A *state function* or *state predicate* is a transition function with no free primed flexible variables. The definitions of free and bound occurrences of variables from Var_e are the usual ones from first-order logic. The semantics of transition functions is defined in terms of a first-order interpretation for the underlying language $\mathcal{L}_P$ that is given in the following.

State formulas

A state of a system consists of a collection Val of *values* that are necessary to describe the system, whereby the degree of abstraction determines which values are necessary. The values include ordinary numbers or strings and the system shall change the values according to an algorithm. To express the values and formulas manipulating them we consider the infinite set Var of variable names and a semantic meaning $[\![M]\!]$ to each syntactic object M. A

[1] http://research.microsoft.com/users/lamport/tla/tla.html (June 2007)

state s is the mapping $s : Var \rightarrow Val$ and thus, the assignment of a value $s(v)$ to the variable v denoted as $s[\![v]\!]$. Flexible variables Var_f are interpreted in a state dependent way, whereby the value of rigid variables Var_r are state independent. They can be seen as constants. The flexible variables are of special interest when specifying a system because they may change their values from state to state via a *state function*. Formally we say for a given first-order interpretation $\mathcal{I}$ of the symbols and an interpretation α of the rigid variables:

$s[\![x]\!]_\alpha = \alpha(v) \quad \text{if} \quad x \in Var_r;$

$s[\![v]\!]_\alpha = s(v) \quad \text{if} \quad v \in Var_f;$

$s[\![f(t_1, \ldots, t_n)]\!]_\alpha = f^{\mathcal{I}}(s[\![t_1]\!]_\alpha, \ldots, s[\![t_n]\!]_\alpha)$ with t_i is a term.

We say $s[\![f]\!]$ denotes the value that $[\![f]\!]$ assigns to state s. Semantically we define $s[\![f]\!]_\alpha \stackrel{\text{def}}{=} f(\forall\, v : s[\![v]\!]/v)$, where $f(\forall\, v : s[\![v]\!]/v)$ denotes the value obtained from f by substituting $s[\![v]\!]$ for v, for all variables v.

Furthermore, we define *state predicates* that are boolean expressions built from rigid and flexible variables, which are evaluated with respect to a single state. The meaning $s[\![P]\!]$ of a predicate P equals *true* or *false* for every state s. We say a state s *satisfies* a predicate P iff (abbreviating: if and only if) $s[\![P]\!]$ equals true, denoted as $s \models P$ iff $s[\![P]\!] = true$. It is:

$s[\![v_1 = v_2]\!]_\alpha$ is true iff $s[\![v_1]\!]_\alpha$ equals $s[\![v_2]\!]_\alpha$;

$s[\![p(t_1, \ldots, t_n)]\!]_\alpha$ is true iff $p^{\mathcal{I}}(s[\![t_1]\!]_\alpha, \ldots, s[\![t_n]\!]_\alpha)$ is true;

$s[\![\neg P]\!]_\alpha$ is true iff $s[\![P]\!]_\alpha$ is not true;

$s[\![P \wedge Q]\!]_\alpha$ is true iff both $s[\![P]\!]_\alpha$ and $s[\![Q]\!]_\alpha$ are true;

$s[\![\exists x : P]\!]_\alpha$ is true iff $s[\![P]\!]_\beta$ is true for $\beta(y) = \alpha(y)$ with $y \in Var_r \backslash \{x\}$;

$s[\![\exists v : P]\!]_\alpha$ is true iff $t[\![P]\!]_\beta$ is true for $t(w) = s(w)$ with $w \in Var_f \backslash \{v\}$.

With a given interpretation α the state formulas describe a set of states, for instance the initial state or reachable states of a modeled system.

Action formulas

Action formulas are predicate formulas containing primed and unprimed flexible variables. They represent relations between two states, the old state and the new state and hence, are often called *transition formulas*. The primed variables are interpreted in the new state and the unprimed variables refer to the old state. It is:

$$(s,t)[\![x]\!]_\alpha = \alpha(x) \quad \text{if} \quad x \in Var_r;$$

$$(s,t)[\![v]\!]_\alpha = s(v) \quad \text{if} \quad v \in Var_f;$$

$$(s,t)[\![v']\!]_\alpha = t(v) \quad \text{if} \quad v \in Var_f.$$

The semantics of a complex term at a pair of states is defined in accordance to the semantics of state formulas that do not depend on the second state of the pair as before mentioned. With a given interpretation α the action formulas describe a set of pairs of states, for instance the intended state transitions of the system. Furthermore, we define (s,t) to be an *A step* iff $s[\![A]\!]t$ equals true for an action A, whereby each unprimed variable in A is interpreted in the old state $s(v)$ and each primed variable is interpreted in the new state $t(v)$. We say that the pair of states (s,t) *satisfies* action A:

$$s[\![A]\!]t \stackrel{\text{def}}{=} A(\forall\, v' : s[\![v]\!]/v, t[\![v]\!]/v').$$

An action A is said to be *valid* (denoted as $\models A$) iff every step is an *A step* with S is the set of all states:

$$\models A \stackrel{\text{def}}{=} \forall s,t \in S : s[\![A]\!]t.$$

For any action A we define a predicate $Enabled(A)$ for all states $s \in S$ that is true for a state s iff it is possible to take an *A step* starting in state s:

$$s[\![Enabled(A)]\!] \stackrel{\text{def}}{=} \exists\, t \in S : s[\![A]\!]t.$$

We additionally define the abbreviation for an action A and terms $t_1, \ldots, t_n$ without primed variables:

$$[A]_{t_1,\ldots,t_n} \stackrel{\text{def}}{=} A \vee (t_1' = t_1 \wedge \cdots \wedge t_n' = t_n).$$

A pair of states (s,t) satisfies formula $[A]_{t_1,\ldots,t_n}$ if it satisfies action A or the values of terms $t_1, \ldots, t_n$ are left unchanged.

Temporal formulas

According to the hierarchy given at the beginning of the section temporal formulas are built from elementary formulas using boolean operators and the additional unary operator $\Box$ (called *always*). Hence, every state formula P is a temporal formula. If A is an action and $t_1, \ldots, t_n$ are terms without primed variables then $\Box[A]_{t_1,\ldots,t_n}$ is a temporal formula. Moreover, if F is a temporal formula so is $\Box F$ as well as all propositional combinations of temporal formulas.

One of the central means of temporal logic are *behaviors*, which are infinite sequences of states, denoted with $\sigma = \langle s_0, s_1, \ldots \rangle$. We denote the suffix $\langle s_n, s_{n+1}, \ldots \rangle$ of a behavior σ with $\sigma[n \ldots]$. Semantically, it is:

$\sigma[\![P]\!]_\alpha$ is true iff $s_0[\![P]\!]_\alpha$ is true where P is a state formula;

$\sigma[\![\Box[A]_t]\!]_\alpha$ is true iff $(s_n, s_{n+1})[\![[A]_t]\!]_\alpha$ is true for all $n \in \mathbf{N}$;

$\sigma[\![\Box\, F]\!]_\alpha$ is true iff $\sigma[n\ldots][\![F]\!]_\alpha$ is true for all $n \in \mathbf{N}$;

$\sigma[\![\neg F]\!]_\alpha$ is true iff $\sigma[\![F]\!]_\alpha$ is not true;

$\sigma[\![\neg F \wedge G]\!]_\alpha$ is true iff both $\sigma[\![F]\!]_\alpha$ and $\sigma[\![G]\!]_\alpha$ are true;

$\sigma[\![\exists x : F]\!]_\alpha$ is true iff $\sigma[\![F]\!]_\beta$ is true with $\alpha(y) = \beta(y)$ for all $y \in Var_r \backslash \{x\}$.

Since the quantification over rigid variables is the one for first-order logic, the quantification over flexible variables is more complex and given in the next section. We additionally define the operator *eventually*: $\Diamond F \stackrel{\text{def}}{=} \neg\Box\neg F$ with the meaning $\sigma[\![\Diamond F]\!]_\alpha$ is true iff $\sigma[n\ldots][\![F]\!]_\alpha$ is true for one $n \in \mathbf{N}$. We will use this operator in the context of fairness conditions in the system specification of the simple smart card model.

System specifications

The specification of a reactive system in TLA is typically a temporal formula of the form

$$Init \wedge \Box[Next]_{\bar{v},\bar{o}} \wedge Fair \quad \text{or in canonical form}$$

$$Init \wedge \Box(A_1 \vee \cdots \vee A_n \vee \bar{v} = \bar{v}' \wedge \bar{o} = \bar{o}') \wedge Fair \quad \text{, where}$$

Init is a state formula describing the initial system state;

Next is the disjunction of action formulas describing the intended state transitions. An action formula A_j holds at least one unprimed and primed flexible variable out of v, o or i;

$\bar{v}$ is a tuple of internal flexible variables $v_1 \ldots v_n$ that may be changed by the system if an action is taken;

$\bar{o}$ is a tuple of flexible variables $o_1 \ldots o_m$ that describe the output of the system and may be changed by the system if an action is taken;

$\bar{i}$ is a tuple of flexible variables $i_1 \ldots i_l$ that describe the input of the system. These variables do not occur in the *stuttering index* of $\Box[Next]_{\bar{v},\bar{o}}$, which allows for arbitrary changes of the input variables. In this way a step of the environment is modeled.

Fair is the conjunction of *fairness* conditions explained below.

The temporal formula of a system specification addresses two powerful concepts implemented in TLA: *fairness* and *stuttering*. A behavior is said to respect the *weak fairness* condition $WF(A)$ for an action A if the action can be taken infinitely often after A has been enabled once upon the time. The *strong fairness* condition $SF(A)$ expresses that A can be taken infinitely often after A has been enabled infinitely often. It is:

$$WF_t(A) \stackrel{\text{def}}{=} \Diamond\Box Enabled\langle A\rangle_t \rightarrow \Box\Diamond\langle A\rangle_t \,;$$

$$SF_t(A) \stackrel{\text{def}}{=} \Box\Diamond Enabled\langle A\rangle_t \rightarrow \Box\Diamond\langle A\rangle_t \,.$$

Fairness conditions can guarantee that particular actions will eventually be taken and hence are the basis for the verification of *liveness properties*. In security analysis we are primarily concerned with *safety properties* that we explain in the next section.

Fairness conditions become especially important when specifying systems that are divided into subsystems called components. In TLA those components may act concurrently and communicate via common flexible variables representing input and output channels. If the behavior of components A and B is given in temporal formulas $Spec_A$ and $Spec_B$ then the behavior of the composed system is given by $Spec_A \wedge Spec_B$, whereby all possible interleavings of all actions have to be considered. We now have to define what a step of the composed system means. Informally we say that at each point in time only one component is allowed to perform a step. All other components do a *stuttering step* that leaves all variables defined in the *stuttering index* unchanged.

Therefore, we define a relation on behaviors such that any two behaviors $p \circ \langle t, t\rangle \circ \sigma$ and $p \circ \langle t\rangle \circ \sigma$ are *stuttering equivalent*, where $\circ$ is the concatenation of behaviors. Intuitively, we replace all sub-behaviors of sequences of the same state t with the single state t in a behavior σ, denoted as $\natural(\sigma)$. We say two behaviors σ and τ are *stutter equivalent* $\sigma \sim \tau$ if $\natural(\sigma) = \natural(\tau)$. In this way we define the behavior of the composed system that combines sequences of states of all components. Furthermore, adding or removing stuttering steps of a behavior σ with $\tau = \natural(\sigma)$ should not affect the truth of a particular temporal formula F:

$\sigma[\![F]\!]_\alpha$ is true iff $\tau[\![F]\!]_\alpha$ is true with $\tau \sim \sigma$.

A proof of this lemma can be found in (AM96). Because all TLA formulas can not distinguish between stutter equivalent behaviors, this property makes refinement via logic implication and the composition of components via logic conjunction of TLA specifications possible.

With the definition of stutter equivalence we can define the quantification over flexible variables, denoted as $\sigma[\![\exists x : F]\!]$:

$\sigma[\![\exists x : F]\!]_\alpha$ is true iff $\tau[\![F]\!]_\alpha$ is true for all $\natural(\sigma) =_x \natural(\tau)$ for all $y \in Var_f \backslash \{x\}$.

Intuitively it means that there is some way of choosing a sequence of values for x such that F holds. The main difference is that there exists at least a value for x in every state of a behavior. The term $\natural(\sigma) =_x \natural(\tau)$ states that two behaviors, which contain no stuttering steps, are equal up to a common state $s_i = t_i$ related to x with $s_i \in \sigma$ and $t_i \in \tau$. The quantification of a flexible variable x leads to the *hiding* of the variable and hence internal structures to the environment. This allows both the description of components in terms of the external behavior only, e.g. input and output variables, and the refinement of the component's internal structure in a further development step. The corresponding quantor $\forall$ is defined as $\forall x : F \stackrel{\text{def}}{=} \neg\exists x : \neg F$.

7.1.2 System verification

TLA provides not only good means for specifying reactive systems but deduction rules for verifying system properties. As in many other formal approaches we are concerned with two main correctness proofs, which are proving that a system specification fulfills specific requirements called properties and that a system implementation is a refinement of the system specification. In the next section we will describe how this can be achieved in TLA.

Proving properties

To prove that a system respects a particular property we represent both the system specification and the property as TLA formulas. A temporal formula F is said to be *valid*, denoted as $\models F$, iff it is satisfied by all behaviors $\sigma \in S^\infty$, where S^∞ denotes all infinite sequences of elements of S:

$$\models F \stackrel{\text{def}}{=} \forall \sigma \in S^\infty : \sigma[\![F]\!].$$

A system specification *Spec* is valid if $\sigma[\![Spec]\!]$ equals true for all behaviors. Additionally, we want a system specification to satisfy a system property *Prop*, which leads to a proof obligation of the form

$$Spec \rightarrow Prop.$$

It holds that $Spec \rightarrow Prop$ is valid iff $\sigma[\![Spec \rightarrow Prop]\!]$ equals true for all behaviors σ.[2] Since $\sigma[\![Spec \rightarrow Prop]\!]$ equals $\sigma[\![Spec]\!] \rightarrow \sigma[\![Prop]\!]$ we say that

[2] A proof of this lemma can be found in (AM96).

every behavior that represents a possible execution of the system *Spec* also satisfies the property *Prop*.

Those properties *Prop* are typically expressed as system invariants of the form $\Box P$ of a state formula P. We already defined that a behavior $\sigma = \langle s_0, s_1, \ldots \rangle$ satisfies the state formula $\sigma[\![P]\!]_\alpha$ iff it satisfies the initial state $s_0[\![P]\!]_\alpha$ of the behavior. We extend this definition by saying that a behavior $\sigma = \langle s_0, s_1, \ldots \rangle$ satisfies the temporal formula $\Box P$ iff P is satisfied in all states of the behavior σ:

$$\sigma[\![\Box P]\!] \stackrel{\text{def}}{=} \forall n \in \mathbf{N} : s_n[\![P]\!] \text{ with } \sigma = \langle s_0, s_1, \ldots \rangle .$$

We will refer to properties of this kind as *safety properties*. Intuitively, safety properties assert that "something bad does not happen"; typical examples include properties of partial correctness, deadlock freedom or mutual exclusion. In security analysis of a system specification we are mainly concerned with properties expressing that a particular action has always been performed before another action is taken. As an example, one may think of the encryption of data before the data are sent via the output channel. The resulting proof obligation is to show that there is no behavior sending unencrypted data via the output channel. Another example taken from operating system design is that every behavior of the operating system successfully checks the access matrix before granting a process access to a data object or resource.

The other well-known class of properties are *liveness properties*, intuitively saying that "something eventually happens". The corresponding TLA property is typically of the form $\Diamond P$. Proving those properties makes the inclusion of fairness conditions necessary. In security analysis we are primarily concerned with safety properties but liveness properties become very import if so-called *denial of service attacks* are considered.

The TLA contains not only syntax and semantics, but rules for proving the mentioned properties. The basic rule for proving an safety property P of a TLA specifications is:

$$\frac{P \wedge Next \rightarrow P' \qquad P \wedge v' = v \rightarrow P'}{P \wedge \Box[Next]_v \rightarrow \Box P}$$

Generally, a proof rule of the form $\frac{F \quad G}{H}$ states that $\vdash F$ and $\vdash G$ implies formula H. A proof rule with no hypotheses is called an axiom. The above mentioned *invariant rule* says that from the validity of the transition formulas $P \wedge Next \rightarrow P'$ and $P \wedge v' = v \rightarrow P'$ we can deduce the validity of the temporal formula $P \wedge \Box[Next]_v \rightarrow \Box P$, where P is a state formula, *Next* is an action and v is a flexible variable. The formula P' is obtained from P by priming all flexible variables.

In practice the formula P is usually proved with respect to a system specification by proving a stronger inductive invariant formula Q with the *inductive invariant rule*:

$$\frac{Init \rightarrow Q \qquad Q \wedge [Next]_v \rightarrow Q' \qquad Q \rightarrow P}{Init \wedge \Box[Next]_v \rightarrow \Box P}$$

Unfortunately, finding proper invariants Q is a creative process, while once Q is found the proof is done schematically. The main advantage here is that most of the work of proving a safety property is done in propositional or first-order logic; at least the deduction of $Init \wedge \Box[Next]_v \rightarrow \Box P$ requires temporal reasoning. In (Lam94; AM96) a complete list of TLA axioms and rules is given for reasoning about TLA formulas. With the given soundness of the rules any derivable formula is valid, i.e. $\vdash F$ implies $\models F$ for any temporal formula F.

Specification refinement

We already discussed TLA in the context of system specification and system property verification. A further characteristic of TLA is that a refinement or an implementation *Impl* of a (verified) specification *Spec* can be expressed as a logical implication, which leads to a prove obligation of a temporal formula of the form:

$$Impl \rightarrow Spec.$$

A temporal formula *Impl* is a *refinement* of a temporal formula *Spec* if $\sigma[\![Impl \rightarrow Spec]\!]$ is valid for all possible behaviors $\sigma \in S^\infty$. In other words, each behavior that satisfies *Impl* has to satisfy *Spec*. Because TLA formulas are stutter invariant (it is they allow steps that leave system variables unchanged) we may add new states and state transitions in the refined specification (a temporal formula) that are not visible in the more abstract specification. The proof strategy of the formula $Impl \rightarrow Spec$ is as follows; we assume:

$$Impl \stackrel{\text{def}}{=} Init^{Im} \wedge \Box[Next^{Im}]_u \wedge Fair^{Im}$$

$$Spec \stackrel{\text{def}}{=} Init^{Sp} \wedge \Box[Next^{Sp}]_v \wedge Fair^{Sp}\,.$$

The proof can be performed in four steps:

1. find and prove a stronger invariant *Inv*: $Impl \rightarrow \Box Inv$;
2. prove the initial condition: $Init^{Im} \rightarrow Init^{Sp}$;

3. prove the simulation condition: $Inv \wedge [Next^{Im}]_u \rightarrow [Next^{Sp}]_v$;
4. prove the liveness condition: $\Box Inv \wedge \Box[Next^{Im}]_u \wedge Fair^{Im} \rightarrow Fair^{Sp}$.

Again, most work of the proof has to be done in ordinary mathematics, whereby the last step requires temporal reasoning. The challenge here is to find a proper state function that maps quantified flexible variable from the temporal formula *Impl* to the temporal formula *Spec*. Those state functions are called *refinement mappings* that give a value for the quantified flexible variable x in formula *Impl* in terms of the quantified flexible variable y in formula *Spec*. Finding proper refinement mappings may require adding *auxiliary variables* such as *history variables* or *prophecy variables* (AL91). This reduces the proof $G \rightarrow (\exists x : F)$ to a proof $G_{aux} \rightarrow (\exists x : F)$, where G_{aux} is the temporal formula G plus the auxiliary variable and x is assumed not to occur free in F. This makes the additional proof necessary that G_{aux} is equivalent to G.

Informally, auxiliary variables are added to a temporal formula without affecting the behavior of the formula. Moreover, those variables will not be implemented in a further refinement step. A formal definition of history variables taken from (Roc04) is: A variable h is a history variable for formula F iff the formula $F \rightarrow Hist(h, f, g, v)$ is valid with
$Hist(h, f, g, v) \stackrel{\text{def}}{=} (h = f) \wedge \Box[(h' = g) \wedge (v' \neq v)]_{\langle (h,v) \rangle}$, where f and v are state formulas, g is an action, h does not occur freely in f or in v and h' does not occur freely in g .

The mentioned history variables are not only helpful for refinement proofs *Impl* $\rightarrow$ *Spec* but for proving safety properties *Spec* $\rightarrow$ *Prop*, whereby those history variables are not necessarily the same in the two proof goals. We will use them in the example of a simple smart card operating system in the next section.

7.2 A smart card system model

In Section 5.2 on Page 44 we discuss the integration of external applications and devices into the smart card operating system. We introduce a new communication component `CommC` that controls all communication between the on-card applications and the outside world. It acts as a gateway between them and ensures confidentiality or integrity of the transferred data if demanded by one of the applications. The abstract scenario is illustrated in Figure 5.7.

According to the proposed scenario, in this section we develop an abstract *Simple Smart Card Model (SSCM)* that concentrates on the external commu-

nication. We skip the `AccessCtrl` component here due to simplicity because we do not need it for the modeling of the external communication. As a first step we identify the subjects of the system model, their communication channels, the objects used in the model and the security objectives to be fulfilled. We then formalize both in TLA the system model and the security objectives of the system in terms of safety properties. Finally, we prove that the system model satisfies the properties. The proofs are performed in the interactive theorem prover VSE-II.

7.2.1 System specification

We identify three components of the Simple Smart Card Model. The Key-Generation Component `KeyGen` is the module that generates key data and stores them in a buffer. A second module, the Communication Component `CommC`, serves as the interface of the smart card. Upon request by an external application it gets key data from the buffer and sends it to the application. The application is obviously not part of the smart card but determines the third entity, the External Application `ExtAppl`. The overall scenario is illustrated in Figure 7.1.

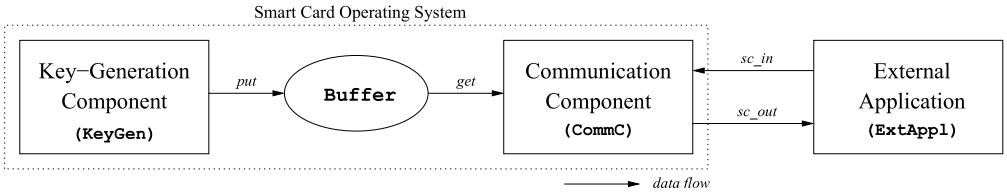


Figure 7.1: A simple smart card system model

Because the generation of keys may take time in the smart card due to limited processing power, we want allow the smart card the stocking of keys. In this way the smart card maintains a pool of keys that can be used upon request. Therefore, we insert a `Buffer` between components `KeyGen` and `CommC`. We want the key generation component to have write access only to the buffer via line `put` as well as the communication component to have read access only via line `get`. We do not specify the buffer in more detail here, because we are not concerned with the design of a FIFO buffer at this abstract level. In fact, the illustrative *Producer-Channel-Consumer (PCC)* example proposed in (RSW+99) can easily be integrated in the design in a further refinement step of the overall engineering process.

The PCC example is concerned with the communication of two arbitrary components (a Producer and a Consumer) via a buffered FIFO-Channel with a finite capacity. The Producer generates arbitrary values and sends them to

the Channel component. From time to time the Channel transmits the values from the internal buffer to the Consumer component according to the FIFO principle. The Consumer receives the values to use them in some further computation. The components have been designed as separate entities that work concurrently, which made the specification of a handshaking protocol necessary. For the implemented communication protocol it has been verified in the tool VSE that no data get lost and that eventually the Consumer receives the values.

In the Simple Smart Card Model we will pay more attention to the interface of the smart card to the outside world, which is given by the input channel `sc_in` and the output channel `sc_out`. Because these channels are assumed to be part of an open network we are confronted with security aspects in the delivering process. That is why we have to take the external application into account in the designing and analyzing process. Therefore, the smart card has to ensure that

- quality proved key data is delivered only (*Quality of Key Data, QK*),
- no key data is delivered twice (*Non-Duplicating, ND*),
- the key data are handled confidentially from the generation to the storing at the External Application (*Confidentiality, CO*) and
- the key data have not been altered without detection from the generation to the storing at the user (*Integrity, IN*).

We will formally verify that the abstract design of the simple smart card operating system fulfills the requirements. In our system model we are not concerned with security protocols, because we assume the underlying protocols to fulfill certain properties given in the next section. In Section 9.2 we will discharge the assumptions made in the cryptographic protocol analysis and made in the system model analysis by combining both approaches.

In the next paragraph we describe the general design of the simple smart card operating system in a more formal way. Therefore, we first define the functions, attributes and predicates needed for modeling the components and the entire system.

Attributes and functions

In the following we refer to two base sets, the set K of *Key Data* and the set A of *Authentication Data*. We assume that the set of correct keys in K is extremely sparse, i.e. a randomly chosen string is a correct key with negligible small probability.

Key Data Key data are generated sequentially $\{k_0, k_1, k_2, \ldots\}$, such that arbitrary k_i, k_j with $i \neq j$ are different $k_i \neq k_j$. We do not specify the key structure in detail because this is unique to a specific cryptographic algorithm. The key data is a string holding all components of the cryptographic key as well as some authentication data used by the predicate p introduced below.

Authentication Data: We assume that at every time each external application `ExtAppl` c can compute one or more authentication token $a \in A$, that allows to identify the external application c unambiguously. It will be transferred to the communication component over an unsecured channel.

Validation of an Authentication: We assume the communication component `CommC` to have a validation function v only known to `CommC` and defined on the set A of authentication data: $v : A \rightarrow \{true, false\}$. This function is used to check, whether a given authentication data a is valid or not. We do not define the details of the validation function because it is not relevant for our model. It may be reasonable that this function has also a time argument, such that a once validated token can not be used anymore. But we will not formalize this and assume that it is already covered by the validation function v.

Integrity Check: We assume, for any key data $k \in K$ a predicate $p(k)$ can be computed in order to check the integrity of all generated keys. If the predicate $p(k)$ is *true*, then the key k was generated by the key generator and vice versa, whereas p is assumed to be public. Additionally, this implicitly proofs the origin and quality of the key data because the key generation component never generates weak keys.

Secure Channel: Given an authentication token $a = a(c)$ of a an external application c with $v\,(a(c)) = true$ the component `CommC` can define a bijection $e : K \rightarrow K$ on the set of key data. We denote this mapping with T: $e = T(a, c) \quad \text{if} \quad v(a(c)) = true$.

The bijective mapping e is known to component `CommC` only, but we will assume that it can be inverted by the corresponding external application only. For the mapping T exists a mapping $T^\star$ so that for the authentication token a and the external application c another arbitrary bijection d' on K can be computed such that:

Let $e = T(a, c)$ and $d' = T^\star(a', c')$ be for an arbitrary authentication token a' of an arbitrary Consumer c' then $T^\star(a', c')\Big(T(a, c)\Big(k\Big)\Big) =$

$d'\Big(e(k)\Big) = k$ holds for a key $k \in K$ if and only if $a' = a$ and $c' = c$ (or $d' = d$).

This defines the transformation of the communication channel between the components `CommC` and `ExtAppl`. For the security of the channel we will further assume that T is designed such that e is known only to `CommC` and $d = T^{\star}(a, c)$ is known only to the legitimate `ExtAppl`. This fact implies that the bijection $e = T(a, c)$ is a sufficiently strong pseudo-random function. As a consequence following from the sparsity of the correct keys it holds for all $k \in K$ that $p(k) = \mathit{true}$ implies $p(e(k)) = \mathit{false}$ because $e(k)$ is a random element of K.

Components of the model

We already introduced the general design of the simple smart card system model and identified subjects and communication channels. With the help of the given attributes and functions we can describe the components and their relations in more detail as illustrated in Figure 7.2.

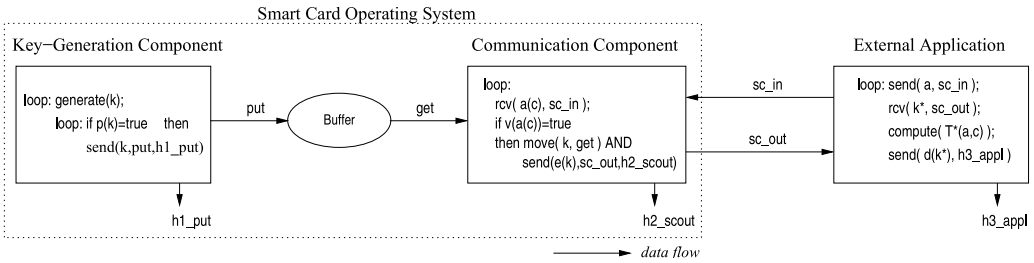


Figure 7.2: Refined simple smart card system model

Key-Generation Component The Key-Generation Component generates arbitrary key data $k_i \in K$ with the help of a hardware random source that is equipped with a live-check to guarantee a sufficient entropy of the generated key data. The component `KeyGen` sends the key data to the `Buffer` via the channel `put` only if the key data satisfy the quality criteria.

Those quality criteria are defined by the requirements of a particular cryptographic algorithm, e.g. requirements of the key length and prime factors in an RSA scheme or requirements on the randomness of an AES key. Because we will not restrict the Key-Generator Component to a specific cryptographic algorithm we say that the criteria check must successfully be passed for a particular scheme. With the positive criteria check `KeyGen` completes the key data k_i with a signature that can only be created by the Producer and hence, leads to the positive value of the predicate $p(k_i) = \mathit{true}$.

We do not specify the buffer in more detail in the model because it would unnecessarily expand our model. In the formal model we will realize the `Buffer` as a list, whereby component `KeyGen` adds an element at the beginning of the list and `CommC` removes an element from the end of the list. In this fashion the Key-Generation Component writes a sequence of key data $k_1, k_2, \ldots$ with $k_i \in K$ in an infinite FIFO buffer (*first-in first-out*). In a further refinement step of the overall engineering process the buffer must be replaced by a component `FIFO-Buffer` as it is given by the PCC model.

In our model we additionally define *history variables* that allow us to gather partial information of the system behavior. History variables provide good means for verification purposes that do not influence the behavior of the component and they will not be implemented. We use them to formulate the invariant properties of the system in Section 7.2.2. The history variable `h1_put` of type list holds all data sent via the channel `put`. The sending of the key data via the channel `put` and `h1_put` is expressed in the function $send(k, \texttt{put}, \texttt{h1\_put})$. Furthermore, it holds that for all $k \in$ `h1_put` is $p(k) = true$ because only successfully checked and correctly signed key data are delivered by the Key-Generation Component. Moreover, if we assume the signature unique to component `KeyGen` it additionally holds for all $\tilde{k} \notin$ `h1_put` is $p(\tilde{k}) = false$. In other words, whenever the predicate p yields true for particular key data, this key data have been generated by the particular Key-Generation Component.

Communication Component The Communication Component accepts *key requests* from an External Application $c \in C$ on the channel `sc_in`, whereas the request holds authentication data $a \in A$. This is modeled in function $rcv(a(c), \texttt{sc\_in})$. Only if the predicate v of component `CommC` yields true $v(a(c)) = true$ for a particular $c \in C$ and $a \in A$ then the Communication Component first gets new key data from `Buffer` via channel `get` and second deletes the key data in the buffer. Please note, component `CommC` can not write in the buffer.

After getting the key data k the Communication Component transforms the data with function e and delivers the transformed data via output channel `sc_out`. This is expressed in function $send(e(k), \texttt{sc\_out}, \texttt{h2\_scout})$. We again define the additional history variable `h2_del` of type list that holds all data sent via the channel `sc_out`.

External Application An arbitrary External Application may send a *key request* to the Communication Component via channel `sc_in`. The component `CommC` will answer the request with a transformed key $e(k)$ via channel

sc_out only if the External Application $c \in C$ sent valid authentication data $a \in A$ with functions $send(a, \texttt{sc\_in})$ and $rcv(k^\star, \texttt{sc\_out})$ with $k^\star = e(k)$.

The External Application c can determine the inverse function d of e only with the help of $T^\star$ that itself calculates from the authentication data a. Hence, only the sender of the authentication data can re-transform the received data $k^\star$ modeled in the function $compute(T^\star(a, c))$. The External Application also requires the definition of a history variable h3_appl of type list that holds all key data received by component ExtAppl given in $send(d(k^\star), \texttt{h3\_appl})$.

7.2.2 Safety properties

In this section we will discuss the security objectives that should be fulfilled by the simple smart card system model. Therefore, we express each objective as a property of the general model given in the previous section, which leads to both *safety properties* and *security properties*. The former are formulated in temporal logic and the latter are considered in a cryptographic protocol analysis. We will first model the properties and then discuss them.

Quality of the key data (QK) The QK-objective states that for each delivered key data k the predicate p validates to *true* every time because only key data that passed the quality check should be delivered by the smart card. This implicitly indicates an evidence of the quality of the key data (see Section 7.2.1). Because the history variable h3_appl stores all delivered key data in a list we have to show that for all delivered key data the predicate p yields true:
$\Box\ (\forall k \in \texttt{h3\_con} : p(k) = true)$.

Non-duplicating Property (ND) The smart card delivers key data at the request of an External Application. At two different requests, whether of the same or different External Applications, it should never deliver key data twice. Hence, a particular key data k should always appear once in the history variable h3_appl:
$\Box\ (\forall k \in \texttt{h3\_con} : k \notin \texttt{h3\_con} \backslash \{k\})$.

Confidentiality (CO) The security objective says that the key data is handled confidentially the entire process chain from the generation in KeyGen to the storing at the External Application ExtAppl. The Simple Smart Card Model is designed to have one interface to the outside world, which are the input channel sc_in and the output channel sc_out. Therefore, we first have to show that no key data appear in the output channel and hence, no key data is an element of the history

variable `h2_scout`. Secondly, we have to show that an attacker is not able to derive information about the delivered transformed key under a specific threat model. We refer to the former property as *safety* and the latter as *security property*, which are discussed below. As long as the authenticated External Application does not self-compromise the received key data, they are kept confidential. The safety property is expressed as: $\Box\ (\forall k \in \texttt{h3\_con} : k \notin \texttt{h2\_scout})$.

Integrity (IN) The integrity objective claims that each received key data of the External Applications have not been altered. An alteration of the key data k would lead to a *false* validation of the predicate $p(k)$. $\Box\ (\forall k \in \texttt{h3\_con} : p(k) = \mathit{true})$.

The simple smart card system model is designed to have two components, the Key-Generation Component `KeyGen` and the Communication Component `CommC`, that we assume to be operated in a secure environment according to a defined security policy, e.g. a tamper-resistant hardware. As discussed in Section 6.1 those security policies distinguish organizational rules and technical (or functional) requirements, whereby the model considered here covers a functional part. Because of the secure environment we need not to consider attack scenarios for the simple smart card system model but design or implementation flaws. The resulting proof obligation is that the system never enters an *insecure state*, whereas insecure is defined by the mentioned properties. In other words, we have to show that *something bad never happens*, which is commonly known as *safety properties*.

In this sense all mentioned properties are safety properties. In order to strengthen the expressiveness of the properties we additionally take the External Applications into account and model all properties in terms of `ExtAppl`. It turns out that both the QK-property and the IN-property can be expressed with the same formula. Please note, the history variable `h3_appl` holds all key data of all users. Furthermore, `h3_con` is modeled as a list (in opposition to a set) that can be extended only, which would let double attached elements appear twice in the list. This is used by the ND-property.

Furthermore, the CO-property is expressed in terms of the predicate p and the history variable `h2_scout`. For the function e defined in Section 7.2.1 holds that $p(k) = \mathit{true}$ implies $p(e(k)) = \mathit{false}$ for arbitrary key data k. Thus, all we state in the CO-property is that the Simple Smart Card Model will never reach a state where non-transformed key data, i.e. in clear text, have been delivered. In other words, the transformation has always been applied on delivered key data.

New proof obligations arise in order to show the *effectiveness* of a chosen transformation T. In this context different attack scenarios have to be con-

sidered because the components `CommC` and `ExtAppl` do not communicate in a secure environment but in an open network. We refer to those properties as *security properties*. In practice, the transformation will usually be instantiated with cryptographic protocols that provide the required characteristics.

Hence, it must first be verified that a particular cryptographic protocol counters particular threats given by a specific threat model, for instance by Dolev and Yao (DY81), and therefore fulfills the requirements of transformation T. Then, it must be verified that the SSCM correctly implements the cryptographic protocol. In other words, we have to show that the communication component behaves exactly as the corresponding agent in the protocol scenario.

Generally, the family of challenge-responds protocols establishing a session key between two parties potentially implements the transformation T. Here, all challenges and responses as well as ID numbers or certificates are represented by the authentication token a and the External Application c. Both `CommC` and `ExtAppl` can compute the secret transformations T and $T^\star$, respectively, with the help of the known (pre-shared) secret and the nonces, whereas T and $T^\star$ represent a specific cryptographic algorithm together with the belonging session key. In Section 9.2 we work out the combination of a verified cryptographic protocol given in Section 8.2 with the simple smart card system model.

7.3 Formalization in VSE-II

In this section we describe the abstract Simple Smart Card Model in the formal specification language VSE-SL (BSI00) and the arising proof obligations within the tool VSE-II (KUW95; HLsS$^+$96; HMR$^+$99). Therefore, we first give a brief overview of the tool and then describe the system formalization an verification. We do not provide a comprehensive introduction into VSE but explain the details of the tool when specifying the SSCM.

7.3.1 Verification Support Environment VSE-II

The *Verification Support Environment (VSE-II)* is a tool for the formal top-down development of structured system specifications and their stepwise refinement towards a high-level programming language, for instance C, using abstract intermediate layers that are also represented by formal specifications. It provides

- a specification language VSE-SL for creating, editing and type checking of system specifications via graphical and textual editors;

- a facility for displaying the development structure called *Development Graph*;
- code generators from VSE-SL to a target language (C, ADA);
- an interactive theorem prover system for treating proof obligations;
- a central data base to store all aspects of the development including proofs and
- an automatic management of dependencies between development steps.

The language definition VSE-SL provides structures for the modeling of state-based systems, which are behaviors (infinite sequences of states), temporal logic formulas, composition of systems and refinements.

Practically, to formalize the Simple Smart Card Model we first have to specify the behavior of every identified component as well as interactions with the environment. We then compose all components to specify the entire system that inherits the behavior of the components according to some predefined composition rules, which is semantically the conjunction of the components. Lastly, we have to specify the properties in a security model, which is itself a state based specification. This way the system specification, i.e. the composed system, must satisfy the security model. The verification task is discussed in more detail in Section 7.3.4.

The central tool in a VSE specification is the *Development Graph* that visualizes all components, their communication channels and the entire system. The Development Graph of the SSCM is given in Figure 7.3, whereby the node `#Definition_Data` represents a subgraph holding the basic definitions of data types, predicates and functions. In the following sections we first give the formalizations of the components `KeyGen`, `CommC` and `ExtAppl`. Then, we compose all components to formalize the entire system and explain necessary fairness conditions to keep the system running. Finally, we sketch the proofs performed in VSE-II.

7.3.2 Formalization of the system components

The system components are specified in temporal logic as a state-based system. The state space of a component is divided into *input lines* holding values from the environment, *output lines* used to deliver values back to the environment, *shared variables* that can be both written and read by defined components and *internal variables* that can accessed by the component itself only and thus, are hidden to the environment. The intended behavior of the component is specified in an *initial state* and possible steps called *actions*.

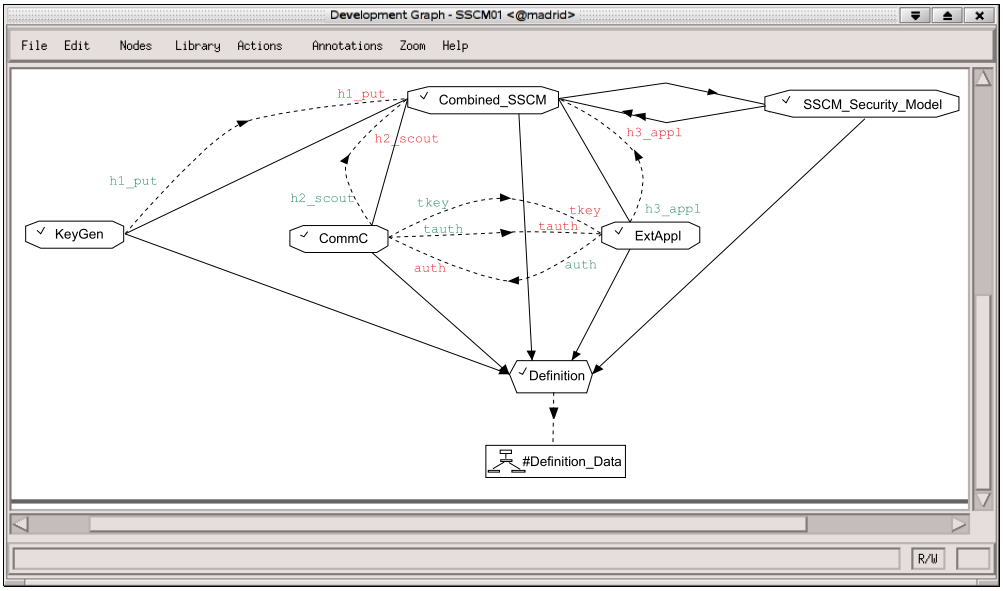


Figure 7.3: VSE Development Graph window of the SSCM

According to the definition of the TLA, the specification of an action contains both preconditions and a description of the resulting state if the action is taken. This is formalized by a predicate formula with primed and unprimed state variables called *flexible variables*. The unprimed state variables denote the system state before the action is taken, and the primed variables denote the state after the action. As described in Section 7.1.1 in TLA action definitions have a particular structure to allow so-called *stuttering* steps. The *stuttering index* mentions those variables of an action that are guaranteed not to change if a stuttering step occurs instead of the action.

From the predicate describing the initial state as well as from the actions one can construct a temporal logic formula in a standard way. Furthermore, the initial state together with the permitted steps describe safety aspects of the component, which can be used to deduce that the system component will never enter undesirable states. This way, modular reasoning within the components helps to analyze the entire system.

The Key-Generation component

The VSE-SL provides a development object of type `TLSPEC` with a number of slots to express the different parts of the specification. In Figure 7.4 the VSE specification window of the component `KeyGen` is given that we describe step by step in the following. The keyword `TLSPEC KeyGen` defines a new object illustrated by a octagon in the Development Graph (see Figure

7.3) that may include already defined abstract data types in the slot `USING Definition` as it is also displayed in the Development Graph by an arrow, whereby abstract data type definitions are visualized in a pentagon. This allows the hierarchical structuring of specifications. The `DATA` slot defines the flexible variables that may change their values from state to state.

```
TLSPEC KeyGen
USING Definition
DATA
   SHARED INOUT key_list : list  /*channel put to buffer*/
   OUT h1_put : list             /*history variable of channel put*/
   INTERNAL xp : nat             /*internal computed random value*/
```

We do not define any input lines for the component `KeyGen` because we assume the hardware random source as part of the Key-Generation Component. There is only one output line to the buffer that is modeled as a shared variable `key_list` of type `list`. Furthermore, we define the output line `h1_put` of type `list` that models the history variable of the output channel. The internal variable `xp` is not accessible by the environment and used for the key data generation.

The `ACTIONS` slot defines the permitted steps of the component, which are a `generate` and a `check_send` action. In our model we abstract from a random source and the key data generation by simply counting numbers. Whenever a new number `xp'` is generated, the flexible variables `key_list` and `h1_put` remain unchanged. Every generated number is checked whether it fulfills the quality requirements summarized in the predicate p and represented by `p_check` or not. The check may sometimes fail or be passed. Only if the test is passed the particular number `xp` representing the key data is added to the list `key_list` representing the buffer and added to the history variable `h1_put`. If this action is taken the variable `xp` remains unchanged. The function `cons(e,l)` append a new element `e` to a list `l`, which denotes the first element of the list.

```
ACTIONS
   generate ::= xp' = xp + 1;
              UNCHANGED(key_list, h1_put)
   check_send ::=
              IF p_check(xp) THEN key_list' = cons(xp, key_list) AND
                                  h1_put' = cons(xp, h1_put) AND
                                  UNCHANGED(xp)
                             ELSE UNCHANGED(xp, key_list, h1_put) FI
```

In the `KeyGen` specification we start counting with 1 in variable `xp` and define the list `key_list` and `h1_put` to be empty in the initial state. In an always

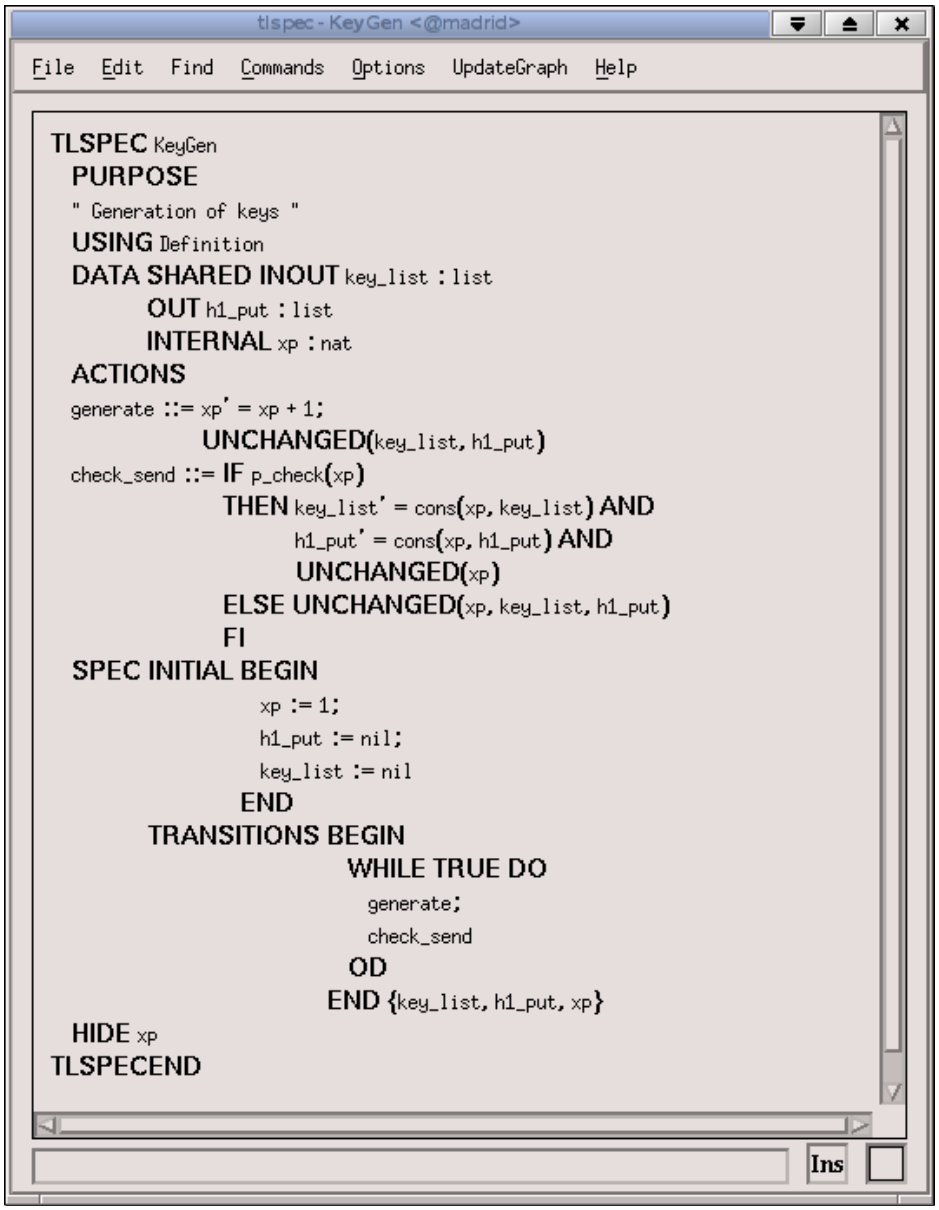


Figure 7.4: VSE specification window of component `KeyGen`

activated WHILE loop KeyGen may generate a number (the key data), checks the quality of the number and in the positive case sends it to the buffer. Otherwise the component performs a stuttering step that let the three variables {key_list, h1_put, xp} unchanged and indicates a step done by the environment. This is modeled in the slot SPEC, whereas the initial state is expressed in INITIAL and the behavior is expressed in TRANSITIONS. Finally, the keyword TLSPECEND closes the VSE-SL specification of the development object.

```
SPEC /* Specification of the behavior of component KeyGen */
   INITIAL     BEGIN
                  xp := 1;
                  h1_put := nil;
                  key_list := nil
               END
   TRANSITIONS BEGIN
                  WHILE TRUE DO
                     generate;
                     check_send OD
               END {key_list, h1_put, xp}
TLSPECEND
```

The VSE-SL allows the specification of behaviors using sequential control flow that is automatically translated into normal actions by introducing a new flexible variable simulating the control flow in the program constructs. For instance, we use the WHILE loop to express that after a generate action is taken the action check_send is activated. When translating the VSE-SL specification into logic formulas by VSE a new flexible variable the_pc of type natural is introduced in the generate and check_send action definition. In the precondition of generate it is set to the_pc=0 and in the postcondition set to the_pc'=1. In the same fashion the check_send action is extended with the precondition the_pc=1 and a postcondition the_pc'=0. In this way an alternating activation of the both actions is realized.

An alternative design without such a control flow could be formalized as follows.

```
   TRANSITIONS [generate, check_send]{key_list, h1_put, xp}
```

This design allows behaviors, where the action generate is taken several times in succession and then the action check_send is activated. This would lead to generated key data that will never be delivered. Moreover, without a modification of the check_send action the KeyGen component would deliver key data twice because the action can also be taken several times in a row. We would have to introduce a variable making sure that the check_send

action can not be taken consecutively. Hence, constructs like `WHILE` loops of the VSE-SL provide convenient means for the design of control flow.

The Communication component

In this paragraph we have a look at the behavior of the Communication Component `CommC`. The complete VSE-SL model of all components as well as the entire system is given in Appendix A. The component `CommC` has an input channel `auth` modeling authentication and thus, key data requests from the external applications via line `sc_in`. The output channel `tkey` models the line `sc_out` to the external applications and delivers the transformed key data. The output channel `h2_scout` models the history variable of channel `tkey` and holds all data delivered via the channel. The additional output channel `tauth` is used to invalidate already used authentication tokens. Lastly, the component `CommC` gets the actual key data from the internal buffer modeled as a shared input and output variable `key_list` that we already introduced in the component `KeyGen`. A visualization of all channels except the shared ones is given in the Development Graph in Figure 7.3.

```
DATA
     SHARED INOUT key_list : list  /*channel get from buffer*/
     IN auth : nat                 /*channel sc_in from ExtAppl*/
     OUT tkey : nat                /*channel sc_out to ExtAppl*/
     OUT tauth : nat               /*actual authentication data*/
     OUT h2_scout : list           /*history variable of sc_out*/
```

The description of the component `CommC` in Figure 7.2 says that it shall wait for authentication requests in order to process the authentication token and to deliver transformed key data. We model all tasks in a single action `checked_transform`. The action gets as input the data `value` representing an actual authentication token. The token is checked for validity and in the positive case together with a positive check for a non-empty buffer of key data the action invalidates the actual authentication token by sending it via channel `tauth`.

Furthermore, the action gets the last key from the buffer `key_list` and transforms it via function `trans(last(key_list))`. The transformed key data are sent via channel `tkey` and are additionally added to the history list `h2_scout` via function `cons(tkey', h2_scout)`. Because by definition of lists the function `last(list)` outputs the last element of the list we additionally need to delete the actual element, which is realized by the postcondition `key_list' = butlast(key_list)`. The function `butlast(list)` outputs the list without the last element of the list. If the preconditions of the action are not fulfilled the variables `(tauth, tkey, key_list, h2_scout)` remain unchanged. Please note, because the function `last(l)` outputs the element

of list `l` and the function `cons(e.l)` adds a new element to the list becoming the first element of `l`, we formalize the buffer according the FIFO principle (First-In-First-Out).

```
ACTIONS
 checked_transform (value : IN nat)::=
     IF v_check(value) AND
        key_list /= nil
     THEN tauth' = value AND
          tkey' = trans(last(key_list)) AND
          key_list' = butlast(key_list) AND
          h2_scout' = cons(tkey', h2_scout)
     ELSE UNCHANGED(tauth, tkey, key_list, h2_scout)
     FI
```

In the initial state of the component `CommC` we set the variables `tauth` and `tkey` to `0` and the history variable `h2_scout` to be empty. In an always activated `WHILE` loop the Communication Component expects new key data requests while waiting for new authentication tokens. If a new token `auth` is received the action `checked_transform(auth)` is performed. If the environment takes a step the variables `{key_list, tkey, tauth, h2_scout}` remain unchanged as given in the stuttering index.

```
 SPEC
      INITIAL BEGIN
               tauth := 0;   tkey := 0;   h2_scout := nil
              END
      TRANSITIONS BEGIN
                   WHILE TRUE DO
                    IF auth /= tauth
                    THEN checked_transform(auth)
                    FI
                   OD
                  END {key_list, tkey, tauth, h2_scout}
```

In this way we formalize in a very abstract way the desired behavior of the Communication Component as described in Section 7.2.1. The resulting proof obligations are discussed in Section 7.3.4.

The External-Applications component

The component External Applications is not part of the smart card but we consider the external applications in order to strengthen the expressiveness of the properties to be proven, which we partly formulate in terms of the component `ExtAppl`.

Therefore, the component ExtAppl has an output variable auth modeling the channel sc_in of the smart card as given in Figure 7.2, which is used to send key data requests to the smart card. The input variable tkey is used to receive the transformed key data and thus, models the output channel sc_out of the smart card. The additional input variable tauth signals invalidated authentication tokens. The variable h3_appl is the history variable of all external applications holding all received and re-transformed key data. The internal variable xc is not visible to the environment and holds the actual re-transformed key data.

```
DATA  INTERNAL xc : nat  /*internal variable not visible to env.*/
      OUT auth : nat     /*send key request via channel sc_in*/
      OUT h3_appl : list /*history variable of received keys*/
      IN tkey : nat      /*receive transformed keys from sc_out*/
      IN tauth : nat     /*CommC current authentication data */
```

We define two actions for the ExtAppl component. The action send_new_req generates new authentication token by counting numbers and the action get_key takes the current transformed key from the input channel tkey, re-transforms it and stores the key data in the internal variable xc. The received key data could now be used for the encrypted transmission of data or to guaranty integrity of the data to be transferred, which is not modeled in the SSCM. The action get_key additionally puts the actual key data in the history list h3_con. The action is not enabled in the initial state of the system because tkey is set to 0 in component CommC initially, which leaves the variables xc, auth, h3_appl unchanged.

```
ACTIONS
send_new_req ::= auth' = auth + 1;
                 UNCHANGED(xc, h3_appl)
get_key ::= IF tkey /= 0
            THEN xc' = inv_trans(tkey) AND
                 h3_appl' = cons(xc', h3_appl) AND
                 UNCHANGED(auth)
            ELSE UNCHANGED(xc, auth, h3_appl)
            FI
```

In the initial state of the component we set the variables auth and xc to 0 and the history list h3_con to be empty. The behavior is formalized in an always activated WHILE loop, where the sending of key requests is always possible via action send_new_req. If the sent authentication token has been invalidated by the smart card, which indicates that the key delivery has been successfully performed, the action get_key can be taken. If the environment takes a step the variables {xc, auth, h3_appl} shall remain unchanged.

```
SPEC INITIAL BEGIN
              auth := 0;
              xc := 0;
              h3_appl := nil
             END
     TRANSITIONS BEGIN
                  WHILE TRUE DO
                   send_new_req;
                   IF tauth = auth
                   THEN get_key
                   FI
                  OD
                 END {xc, auth, h3_appl}
```

The complete specification of the component `ExtAppl` and the entire system is given in Appendix A.

7.3.3 Fairness and concurrency

So far we modeled each component of the SSCM independently that may communicate with other components via input and output channels or via shared variables. The composition of two arbitrary components is technically done in the `COMBINE` slot of a further `TLSPEC` object that defines both connections between the components and connections of the components with the composed system. Additionally, the possible behaviors of the composed system may be further restricted by inserting supplementary axioms. The number of components to be composed is not limited in VSE-II and composed systems themselves can in turn be components of other systems. In this way complex system specifications can be structured. In the Simple Smart Card Model we need to combine the Key-Generation Component `KeyGen`, the Communication Component `CommC` and the External Application `ExtAppl` in the new component `Combined_SSCM`.

The connections between components are defined in the `COMBINE` slot of the object `TLSPEC Combined_SSCM`, whereas the arrow in-between the variables indicates the direction in which data, i.e. information, flows. Shared variables are connected in an extra slot of the `COMBINE` slot. The `DATA` slot defines all connections of the components with the combined system, which are the history variables of each component `h1_put`, `h2_kdel` and `h3_con`. Please note, the history variables will not be implemented in a further refinement step; they are auxiliary variables in order to perform proofs about the system behavior. The invariant properties to be satisfied by each system behavior are modeled in a separate object `SSCM_Security_Model` that we discuss in the next section.

```
TLSPEC Combined_SSCM
 USING Definition
 DATA
      OUT h1_put : list   /*history variable of the channel put*/
      OUT h2_scout : list /*history variable of the channel sc_out*/
      OUT h3_appl : list  /*history variable of all external appl.*/
 COMBINE KeyGen [KeyGen.h1_put -> Combined_SSCM.h1_put] ;
         CommC [CommC.h2_scout -> Combined_SSCM.h2_scout,
                CommC.auth <- ExtAppl.auth]
         SHARED [CommC.key_list <- KeyGen.key_list];
         ExtAppl [ExtAppl.h3_appl -> Combined_SSCM.h3_appl,
                  ExtAppl.tauth <- CommC.tauth,
                  ExtAppl.tkey <- CommC.tkey]
 SATISFIES SSCM_Security_Model
TLSPECEND
```

We already mentioned in Section 7.1.1 that a particular behavior σ is a behavior of the combined system if and only if σ is a behavior of each system component. The concurrent execution of the components `KeyGen`, `CommC` and `ExtAppl` is formalized by logic conjunction `KeyGen` $\wedge$ `CommC` $\wedge$ `ExtAppl` and considers all possible interleavings of all actions. For proving purposes it is necessary to know for each step of a behavior of the entire system which component performed the step. From the local perspective of a component stuttering steps have been introduced to distinguish steps done by the component and done by the environment.

VSE-SL offers the construct of `SHARED INOUT` variables that we use to model the internal buffer. Shared variables generally do not occur in the stuttering index, which makes the introduction of an additional predicate $is\_active(Spec_A)$ for each component $Spec_A$ necessary in order to determine which component performed the step (HMR+99; RSW+99). The specification of a component given by

$$Spec_A = Init \wedge \Box(A_1 \vee \cdots \vee A_n \vee \bar{v} = \bar{v}' \wedge \bar{o} = \bar{o}') \wedge Fair$$

has then to be of the following form for the action formulas:

$$((A_1 \vee \cdots \vee A_n) \wedge is\_active(Spec_A)) \vee (\neg is\_active(Spec_A) \wedge \bar{v} = \bar{v}' \wedge \bar{o} = \bar{o}'),$$

where A_i denotes actions and $\bar{v}$ and $\bar{o}$ are tuple of internal variables and output variables. Input variables and shared variables do not occur in the stuttering index because they may change if the environment performs a step. Hence, the specification of a combined system composed of components $Spec_A$ and $Spec_B$ is given by

$$Spec_A \wedge Spec_B \wedge \Box(\neg is\_active(Spec_A) \vee \neg is\_active(Spec_B)).$$

For VSE-SL specifications the VSE system enlarges the specifications automatically by the *is_active*() predicate when translating the specification into formulas. We will use the *is_active*() predicate in the proofs described in the next section.

In a TLA specification it might be necessary to add fairness conditions in order to achieve the desired behavior of the system component and hence, the entire system. Weak and strong fairness conditions can be explicitly added for every action in the `Spec` slot of a TLA specification `TLSPEC`. We already discussed control flow constructs that also determine the behavior of the system. When the VSE system translates those constructs into sets of actions it automatically adds fairness conditions.

In the example of the `KeyGen` component this leads to four weak fairness conditions. According to the `WHILE` loop construct and the two defined actions four cases are distinguished. Firstly, if the `WHILE` loop is executed the action `generate` can be taken that in turn disables itself and enables the `check_send` action. The firing of the `check_send` action denotes the second case and a behavior where no action of the executed `WHILE` loop fires is the third case. The last case is also no taking of rules because the `WHILE` loop is disabled. The automatically introduced weak fairness conditions guaranty that all behaviors occur infinitely often. These fairness conditions are sufficient to keep our system running. In the proofs we will not use the fairness formulas because we are concerned with safety properties.

7.3.4 Verification of safety properties

While the system model specifies the system, the security model describes the important properties of the system. These properties must be logically implied by the system specification or in other words, the system specification must satisfy the security model.

In VSE-II a formal security model of a state-based system specification is as well a state based system. The inclusion of the system is recommended in order to inherit all state variables and used theories of the system specification. This helps us to express the properties of this system. Including other state-based systems is done by referring to them in the `INCLUDE` slot of a further `TLSPEC` object named `PSC_Security_Model`. The properties are then added in the system behavior, which is in the `SPEC` slot. Because the system specification must satisfy the security model we add in the `TLSPEC Combined_SSCM` the slot `SATISFIES SSCM_Security_Model`, which is visualized by a double arrow in the Development Graph in Figure 7.3.

As already mentioned, in the Simple Smart Card Model we make use of history variables `h3_appl` and `h2_scout` when formulating the properties

to be proven. These variables correspond to other variables `xc` and `tkey`, respectively, whose histories they represent. Here, `xc` is an internal variable of the component `ExtAppl` holding the value of the current re-transformed key data. These lists contain all the values that have been assigned to the corresponding variables in all states from the initial state up to the current state of the behavior of the entire system. Formally, `h3_con` being the history variable for `xc` means:

$$\texttt{h3\_appl} = \texttt{cons}(\texttt{xc}, nil) \wedge$$
$$\Box((\texttt{h3\_appl}' = \texttt{h3\_appl} \wedge \texttt{xc} = \texttt{xc}') \vee \texttt{h3\_appl}' = \texttt{cons}(\texttt{xc}', \texttt{h3\_appl})),$$

where the function `cons(e,l)` appends an element `e` to a list `l`.

The presence of history variables enables us to express conditions in terms of past values of certain state variables. In this way we can express that `xc` does not take the same value twice as required in the ND-property. In the generic security model we define a new predicate `greatest_elem(e,l)` that validates true if an element `e` of type `naturals` is the greatest number of the list `l` of natural numbers. Because the Key-Generator Component is modeled to generate ascending natural numbers we say the predicate `greatest_elem(first(h3_con),rest(h3_con))` must always be true for the first element of `h3_con` being the greatest element of the list `h3_con` without the first element. The function `first(list)` outputs the first element of the list, and the function `rest(list)` outputs the list without the first element of the list. Moreover, by construction of lists the lastly appended list element via function `cons()` is always the first element of the list. Thus, the statement is equivalent to $\Box\ (\forall k \in \texttt{h3\_appl} : k \notin \texttt{h3\_appl} \backslash \{k\})$.

Please note, with the proof of the expression we show that there is no behavior of the entire system delivering key data twice. The proof makes the assumption that no key data is generated twice, which we model by simply counting numbers. In a refinement of the component this action must be replaced by a random number generator, whereby the generator must be verified to meet our assumption that no data is generated twice.

As discussed in the previous section the QK-property and the IN-property can be expressed with the same formula: $\Box\ (\forall k \in \texttt{h3\_appl} : p(k) = \mathit{true})$. In the VSE-II model we say that in every state the predicate p represented by `p_check` yields true for the first element of the list `h3_con` beginning with the empty list in the initial state.

The confidentiality property CO is expressed in the same fashion. By definition of the transformation function `trans(`k`)` it holds for key data k that `p_check`$(k) \rightarrow \neg$`p_check(trans(`k`))`. Hence, we have to prove that for all delivered key data the predicate `p_check` does not hold, which we do by

showing that always the first element of the history list `h2_scout` validates `p_check` to false.

```
TLSPEC SSCM_Security_Model
 PURPOSE
 " Specification of the security model of the entire system "
 USING Definition
 INCLUDE S_S_C_M = Combined_SSCM
 SPEC
   /* confidentiality property: */
   [] (S_S_C_M.h2_scout /= nil ->
       NOT p_check(first(S_S_C_M.h2_scout)));
   /* integrity property: */
   [] (S_S_C_M.h3_appl /= nil ->
       p_check(first(S_S_C_M.h3_appl)));
   /* non-duplicating property: */
   [] (S_S_C_M.h3_appl /= nil ->
       greatest_elem(first(S_S_C_M.h3_appl), rest(S_S_C_M.h3_appl)))
TLSPECEND
```

The VSE automatically generates proof obligations that have to be discharged. In the following we exemplary give the proof sketch of the integrity property. To prove the property for the entire system we need to prove lemmas locally to the components. We first have to show that the `KeyGen` component puts successfully checked key data in the buffer `key_list` only. Therefore, we prove the following lemma locally in the `KeyGen` specification stating that from the perspective of the component `KeyGen` the first element of list `key_list` is always true for the predicate `p_check`.

$$\Box(\texttt{KeyGen.key\_list} \neq nil) \rightarrow \texttt{p\_check}(\texttt{first}(\texttt{KeyGen.key\_list}) \quad (7.1)$$

For convenient purposes we define a predicate `p_check_list(`l`)` over lists that yields true if the list is empty or if for all elements of the list `p_check` is true.

$$\begin{aligned} \texttt{p\_check\_list}(l) \leftrightarrow (l = nil \vee \\ (\texttt{p\_check}(\texttt{first}(l)) \wedge \\ \texttt{p\_check\_list}(\texttt{rest}(l)))) \end{aligned} \quad (7.2)$$

We additionally prove the lemma

$$\Box(\texttt{p\_check\_list}(\texttt{KeyGen.key\_list})). \quad (7.3)$$

The proof of Lemma 7.3 is done locally to the behavior of the component `KeyGen`. As long as `KeyGen` is active and thus, performing steps we can proof

the lemma because there is no behavior of the component, where it may put key data `xp` in the `key_list` without a successful validation of `p_check(xp)`. A problem occurs when the environment is performing a step because from the `KeyGen` perspective we can not say what happens with the shared variable `key_list`. Therefore, we have to make an assumption about the environment saying that only the last element of `key_list` will be removed or `key_list` remains unchanged. This is in fact true because we modeled the `CommC` component in that way. We formulate Assumption 7.4.

$$\neg \texttt{is\_active}(\texttt{KeyGen}) \rightarrow (\texttt{key\_list}' = \texttt{key\_list} \vee \texttt{key\_list}' = \texttt{butlast}(\texttt{key\_list})) \tag{7.4}$$

In general, all knowledge needed about the environment has to be inserted via assumptions and is stored in the lemma base of the component. A proof in VSE-II is represented by a proof tree shown in Figure 7.5 for Lemma 7.3. The colored circles denote an insertion of a lemma and the black circles denote a deduction step by applying a deduction rule. In Figure 7.5 applying the `'case distinction'` rule gives three cases: one if the component `KeyGen` is active, one if it is not active and a third case if nothing happens. The middle black circle denotes the second case that we can close by inserting the Assumption 7.4. All assumptions made have to be discharged in the entire system as we will see later on.

Next, we have a local look at the behavior of the component `CommC` that after a successful validation of an authentication token takes key data from the buffer `key_list`, transforms it via function `trans()` and outputs it via variable `tkey`. We locally prove the following lemma that we use later on in the component `ExtAppl`.

$$\Box(\neg(\texttt{CommC.tkey} = 0) \rightarrow \texttt{p\_check}(\texttt{inv\_trans}(\texttt{CommC.tkey}))) \tag{7.5}$$

Lemma 7.5 states that for re-transformed key data k the predicate `p_check` validates true, which holds by definition of the transformation:

$$\texttt{p\_check}(k) \rightarrow \neg\texttt{p\_check}(\texttt{trans}(k)) \quad \text{and} \quad k = \texttt{inv\_trans}(\texttt{trans}(k)).$$

To perform the proof we need to make the assumption that for all key data taken from the buffer `key_list` the predicate `p_check` holds:

$$\Box(\texttt{p\_check}(\texttt{last}(\texttt{CommC.key\_list}))) \tag{7.6}$$

Next, we show that component `ExtAppl` receives correct key data only, which we describe with the following lemma stating that the first element of history list `h3_appl` validates predicate `p_check` to true:

$$\Box(\texttt{ExtAppl.h3\_appl} \neq \mathit{nil}) \rightarrow \texttt{p\_check}(\texttt{first}(\texttt{ExtAppl.h3\_appl}), \tag{7.7}$$

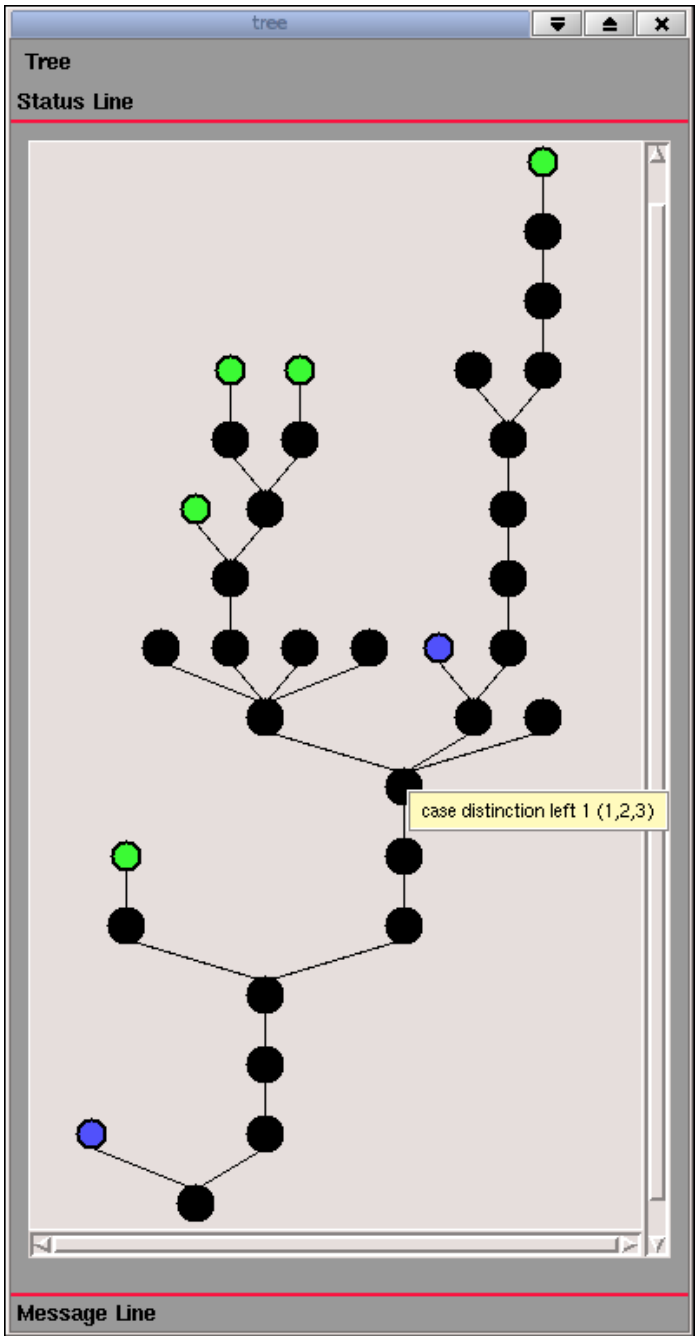


Figure 7.5: VSE proof tree window of Lemma 7.3

which makes the assumption necessary that for all received and correctly re-transformed key data the predicate `p_check` holds.

$$\Box(\neg(\texttt{ExtAppl.tkey} = 0) \rightarrow \texttt{p\_check}(\texttt{inv\_trans}(\texttt{ExtAppl.tkey}))) \quad (7.8)$$

So far, we proved Lemma 7.7 locally for every behavior of component `ExtAppl`. We aim at proving Lemma 7.7 for every behavior of the entire system that is formulated in the security model `SSCM_Security_Model` of the SSCM.

$$\Box(\texttt{S\_S\_C\_M.h3\_appl} \neq nil) \rightarrow \texttt{p\_check}(\texttt{first}(\texttt{S\_S\_C\_M.h3\_appl}) \quad (7.9)$$

We can prove Lemma 7.9 by inserting Lemma 7.7 because the variable `h3_appl` is controlled by the component `ExtAppl` only.

It remains to discharge Assumptions 7.4, 7.5 and 7.8 of the individual components, which become lemmas in the combined system. To prove Lemma 7.4 stating that buffer `key_list` is reduced only or remains unchanged if component `KeyGen` is not active we import the behavior of components `CommC` and `ExtAppl`. If component `ExtAppl` is active the proof is trivial because the variable is hidden for the component and thus, can not be changed by it. If component `CommC` is active we have to prove that each behavior either leaves the variable unchanged or removes the last element via function `butlast()`, which is indeed true. Please note, the entire system is active if and only if exactly one component is active.

$$\begin{aligned}\texttt{is\_active}(\texttt{Combined\_SSCM}) \leftrightarrow (&\texttt{is\_active}(\texttt{KeyGen}) \vee \\ &\texttt{is\_active}(\texttt{CommC}) \vee \\ &\texttt{is\_active}(\texttt{ExtAppl}))\end{aligned} \quad (7.10)$$

Furthermore, Assumption 7.6 saying that the last element of `key_list` is true for `p_check` is also exported to the lemma base of the combined system and can be proved with the help of Lemma 7.3 stating that for all elements of the list `key_list` the predicate `p_check` holds. By definition of lists and `p_check` it holds:

$$\texttt{p\_check\_list}(l) \rightarrow \texttt{p\_check}(\texttt{last}(l)) \quad \text{and}$$

$$\texttt{p\_check\_list}(l) \rightarrow \texttt{p\_check\_list}(\texttt{butlast}(l)).$$

Lastly, Assumption 7.8 also becoming a lemma in the combined system can be proved by inserting Lemma 7.5 from component `CommC`.

It must be pointed out that the technique of making assumptions and discharging them is liable to circular reasoning meaning that two assumptions are mutually dependent and thus, can not be discharged unless one of them can be discharged. A solution to this problem is proposed in (RSW$^+$99) with

the introduction of an `unless` operator available in VSE-II. In the SSCM we are not confronted with circular reasoning.

In this way we verified the properties of the security model and hence, verified the Simple Smart Card Model.

7.4 Conclusion

The extended security policy introduced in Chapter 5 controls both the communication between on-card applications by means of access control mechanisms and the external communication by additional means of cryptographic protocols. Therefore, in a Simple Smart Card Model a new Communication Component is introduced controlling all communications between an on-card application and the outside world. If confidentiality or integrity of the transferred data is required it additionally provides a cryptographic session key, e.g. for encryption purposes. In Section 6.2.1 we identified that the formal analysis of access control mechanisms leads to the formal verification of safety properties, whereas the formal analysis of mechanisms securing the external communication leads to the formal verification of security properties as discussed in Section 6.2.2. A visualization of the Simple Smart Card Model (SSCM) is given in Figure 5.7 in Section 5.2.3 on page 48, where access control mechanisms are denoted in red color and mechanisms controlling external communications are denoted in blue color.

In this chapter we formalize the SSCM that focuses on the three components Key-Generation `KeyGen`, Communication `CommC` and External-Applications `ExtAppl`. We formalize each component as a single state-based system in the Temporal Logic of Actions communicating with other components via input and output channels and via shared variables. We formally verify for each possible behavior of the entire system that the Communication component and thus, the smart card never delivers key data of unproved quality, never delivers key data twice and never delivers key data in clear text. To stress the expressiveness of the proofs we formulate the invariant properties of the entire system in terms of the External-Applications that are not part of the smart card.

It turns out that we in fact proved safety properties, for instance stating that no key data are delivered in an "untransformed" way. In this manner we prove that the transformation function is always applied on key data before delivering. The transformation function itself has well defined properties, e.g. transformed key data can only be re-transformed by the correct corresponding external application. According to our discussion in Section 6.2.3 we did not put an attacker model into account and thus, did not verified

security properties, which we marked in red colored boxes in Figure 6.4.

Hence, the question arises: Did we miss our aim? We just performed the first step by showing that each behavior of the system represented by the SSCM respects the security mechanisms controlling the external communication. In further steps we have to show that

1. an instantiation of the abstract transformation function indeed exists;
2. the found instantiation of the transformation function is correctly implemented by the Communication Component.

We will work out the both issues in the next chapters by first showing that a cryptographic protocol performing a mutual authentication with session generation indeed fulfills the requirements of the transformation function. We then show that the extended Communication component behaves exactly as specified in the verified protocol.

Chapter 8

Security properties of cryptographic protocols

In the previous chapter we specified a simple smart card system modeling the extended security policy introduced in Chapter 5. In a formal state-based model of the system we verified safety properties, for instance stating that no key data are delivered without applying the transformation function T. In the verification we did not take an attacker into account and defined various assumptions for the transformation function, e.g. with a given valid authentication token the corresponding external application only can re-transform delivered key data.

In practice, those functions need to be refined, which is the aim of this chapter. The mentioned transformation T can be instantiated by a cryptographic protocol providing a mutual authentication between two participants and establishing a session key for the confidential communication or for the integrity of the transferred data. We want formally verify that a certain cryptographic protocol indeed fulfills the defined assumptions in the simple smart card model even an attacker with particular skills is involved. Therefore, we first introduce the inductive approach by Paulson and then discuss a certain protocol in order to formalize and verify the protocol.

8.1 Paulson's inductive approach

In the following we introduce the inductive approach to verifying cryptographic protocols (Pau98). Therefore we use a notation for protocol description due to Paulson that include

- agent names in capital letters A, B, ...;

- nonces with capital letter N_a, N_b, ... indexed with the creating agent and guessable number N with no index;
- keys with capital letter K_a, K_b, K_{ab}, ... indexed with the possessing agents;
- compound messages $\{\!|X, X'|\!\}$ with X and X' are arbitrary messages;
- hashed message $\texttt{Hash}\, X$;
- encrypted message $\texttt{Crypt}\, K\, X$.

Hence, a message may either contain agent names, nonces, keys, hashed messages, encrypted messages or a composition of them. As in all other protocol verification techniques the underlying cryptography is assumed to be perfect. Because the hashing shall be collision-free and encryption is strong, it holds

$$\begin{aligned} &\texttt{Hash}\, X = \texttt{Hash}\, X' \quad \text{only if} \quad X = X' \quad \text{and} \\ &\texttt{Crypt}\, K\, X = \texttt{Crypt}\, K'\, X' \quad \Longrightarrow \quad K = K' \wedge X = X'. \end{aligned}$$

Furthermore, a message X can not be altered or extracted from $\texttt{Crypt}\, K\, X$ without first decrypting it with the corresponding key K, which makes the possession of the key K necessary. In a public-key encryption scheme K^{-1} is the inverse of key K with $(K^{-1})^{-1} = K$ for all K. If the equality $K^{-1} = K$ holds, then K is a symmetric key. With the basic notation we can start modeling a protocol.

8.1.1 The basic formalism

In the basic model the agents together with the attacker are allowed to *send*, to *receive* and to *note* a message transferred in the network. Thus, three kinds of *events* are defined:

- event $\texttt{Says}\, A\, B\, X$ means "A attempts to send message X to B";
- event $\texttt{Gets}\, B\, X$ means "B receives message X from the network";
- event $\texttt{Notes}\, A\, X$ means "A stores X internally".

The event $\texttt{Notes}\, A\, \{\!|X, X'|\!\}$ is used to note down a portion of a received message, e.g. X', and is visible to A and additionally to the attacker if A is compromised. Generally, agents only read messages addressed to themselves except the attacker who may read all transferred messages. We model the attacker in more detail later on. In (Bel07) further events are defined for the

handling of time stamps and for the modeling of protocols based on smart cards. At this stage we do not take them into account.

Each step of a protocol has to be described in terms of events ev, and a protocol run is modeled as the extension of a *trace* with new events, called evs. We define inductively:

1. `Nil-Rule`: The empty list $[\,]$ is a trace evs.

2. `Says-Rule`: If evs is a trace and $A \neq B$, then evs may be extended with the event $ev = \texttt{Says}\, A\, B\, X$, denoted as $ev\#evs$.

3. `Gets-Rule`: If evs is a trace and $\texttt{Says}\, A\, B\, X \in evs$, then evs may be extended with the event $ev = \texttt{Gets}\, B\, X$.

4. `Notes-Rule`: If evs is a trace and $\texttt{Says}\, A\, B\, X \in evs$, then evs may be extended with the event $ev = \texttt{Notes}\, B\, X$.

In this way we inductively define the set of possible traces, which is the least set closed under the given *rules*. Thus, a protocol must be modeled in terms of rules, whereby an event occurs via the firing of these rules. For instance, a protocol described in five steps will be modeled in five inductive rules. The firing of rules is non-deterministic, which means if the pre-conditions of several rules hold there is no tactic which of them will be taken. For example, an agent may react on a received message by sending a response or the agent may start a new protocol session by sending the very first message of a protocol run. In this way the *interleaving* of protocol runs is indirectly modeled. Moreover, no rule is forced to fire, which implicitly models the *interception* of messages by the attacker. This, of course, requires *fairness* of all agents because if no agent sends a message (fires a rule) then the analysis of such protocol makes no sense.

In order to prove a property φ in accordance to the inductive principle we must show that it first holds for the empty trace $\varphi([\,])$ (the inductive hypotheses) and then we must prove an assertion of the form $\varphi(evs) \Rightarrow \varphi(ev\#evs)$, where ev is an event according to the protocol rules (the induction step). A trivial example of induction, given in (Pau98), is to prove that no agent sends a message to himself, i.e. no trace contains an event of the form $\texttt{Says}\, A\, A\, X$. This obviously holds for the empty trace. The remaining possible event `Says` specifies a condition $A \neq B$ that prevents the creation of such event and the event `Notes` does not send a message anyway.

So far the entire model of a protocol is trivial. In order to express security conditions we additionally need to model the network, which is done by the modeling of the attacker in the next section.

8.1.2 Protocol verification

We already identified one property of the attacker, which is the interception of messages since no rule is forced to fire. Paulson considers three additional operators `parts`, `analz` and `synth` on possible infinite sets of messages that are defined inductively, as are protocols themselves. The operators extend a set of messages H with other items derivable from H. Generally, H is the initial knowledge of the attacker together with all messages sent in a trace.

The set $\texttt{parts}\, H$ contains H and all messages that can be recursively extracted from H by decomposing compound messages and decrypting messages. It represents the set of all components of H that are *potentially* recoverable. Formally, it is defined to be the least set closed under the following rules, where $\{X, Y\}$ denotes a set with elements X and Y.

$$\frac{X \in H}{X \in \texttt{parts}\, H} \qquad \frac{\texttt{Crypt}\, K\, X \in \texttt{parts}\, H}{X \in \texttt{parts}\, H} \qquad \frac{\{\!|X, Y|\!\} \in \texttt{parts}\, H}{\{X, Y\} \in \texttt{parts}\, H}$$

Here, proving the fact $X \notin \texttt{parts}\, H$ says that X has almost not been transferred in the network, except in hashed form. On the other hand, the fact $X \in \texttt{parts}\, H$ states that X has been transferred whether in clear or in encrypted form and could potentially be decrypted. In contrast, the operator `analz` additionally states that the decryption has been performed. It is the least set including H and closed under projection and decryption with known keys.

$$\frac{X \in H}{X \in \texttt{analz}\, H} \qquad \frac{\texttt{Crypt}\, K\, X \in \texttt{analz}\, H \quad K^{-1} \in \texttt{analz}\, H}{X \in \texttt{analz}\, H}$$

$$\frac{\{\!|X, Y|\!\} \in \texttt{analz}\, H}{X \in \texttt{analz}\, H} \qquad \frac{\{\!|X, Y|\!\} \in \texttt{analz}\, H}{Y \in \texttt{analz}\, H}$$

This operator can be used to express the fact that a key $K \notin \texttt{analz}\, H$ is not obtainable by listening the network, i.e. listening to H. Finally, the set $\texttt{synth}\, H$ models messages that can be created by the attacker. This includes the adding of agent names and guessable numbers (no nonces), the hashing and composition of messages as well as encryption of messages with keys already known to the attacker.

$$\begin{array}{c}\texttt{Agent}\, A \in \texttt{synth}\, H \\ \texttt{Number}\, N \in \texttt{synth}\, H\end{array} \qquad \frac{X \in H}{X \in \texttt{synth}\, H} \qquad \frac{X \in \texttt{synth}\, H}{\texttt{Hash}\, X \in \texttt{synth}\, H}$$

$$\frac{X \in \texttt{synth}\, H \quad Y \in \texttt{synth}\, H}{\{\!|X, Y|\!\} \in \texttt{synth}\, H} \qquad \frac{X \in \texttt{synth}\, H \quad K \in H}{\texttt{Crypt}\, K\, X \in \texttt{synth}\, H}$$

With the three operators we can define two additional rules for new events on order to strengthen the abilities of the attacker. The rules shall model *fake* messages and *accidents*. We define inductively:

5. `Fake-Rule`: If evs is a trace and $X \in \texttt{synth}(\texttt{analz}\, H)$ is a fraudulent message and $Spy \neq B$, then evs may be extended with the event $ev = \texttt{Says}\, Spy\, B\, X$.

6. `Oops-Rule`: If evs is a trace and $\texttt{Says}\, A\, B\, \texttt{Crypt}\, K \{\!|N_a, K'|\!\} \in evs$, then evs may be extended with the event $ev = \texttt{Notes}\, Spy\, \{\!|N_a, K'|\!\}$.

The *Fake-Rule* states that the attacker may say anything she can generate from past messages and from her initial knowledge. This knowledge may include shared-keys or private keys of an arbitrary set of compromised agents. In this case she can masquerade as any of the so-called *bad* agents. The *Oops-Rule* models the accidental loss, however, of a session key or shared key. This rule helps proving that a lost key can not compromise future runs of the protocol. Thus, the rule may include nonces to distinguish between recent and past losses.

To reason about the knowledge of the attacker and hence the security of the protocol we need to model the knowledge of the agents and the attacker in a trace. Therefore, in the empty trace the initial knowledge of every agent are the shared-keys `shrK` with other *friendly* agents. On may also think of private keys if asymmetric cryptography is used as proposed in (BP06), but we omit this here. In protocols using symmetric cryptography very often a trusted server, say S, is used that initially holds long-term keys of every participating agent. Furthermore, in the empty trace the attacker holds all shared keys of compromised agents. The initial knowledge is defined as:

$$\begin{aligned} \texttt{initState}\, S &\stackrel{\text{def}}{=} \text{all long term keys} \\ \texttt{initState(Friend}\, i\texttt{)} &\stackrel{\text{def}}{=} \{\texttt{Key(shrK(Friend}\, i\texttt{))}\} \\ \texttt{initState}\, Spy &\stackrel{\text{def}}{=} \{\texttt{Key(shrK(A))} | A \in \texttt{bad}\}. \end{aligned}$$

With the extension of a trace with new events the attacker may expand her knowledge with messages she can see in an event. In this way the view of the attacker on the network traffic is modeled. The function `spies` records all traffic of the entire network in the knowledge base of the attacker as well as the internal notes of the compromised agents.

$$\begin{aligned} \texttt{spies}\, [\,] &\stackrel{\text{def}}{=} \texttt{initState}\, Spy \\ \texttt{spies}((\texttt{Says}\, A\, B\, X)\#evs) &\stackrel{\text{def}}{=} \{X\} \cup \texttt{spies}\, evs \\ \texttt{spies}((\texttt{Notes}\, A\, X)\#evs) &\stackrel{\text{def}}{=} \begin{cases} \{X\} \cup \texttt{spies}\, evs & \text{if } A \in \texttt{bad} \\ \texttt{spies}\, evs & \text{otherwise} \end{cases} \end{aligned}$$

Furthermore, the function `used` defined on traces formalizes the notion of *freshness*. A message X is considered to be fresh if it is neither in the initial

knowledge of every agent nor a component of a sent message of the trace.

$$\texttt{used}\,[\,] \stackrel{\text{def}}{=} \bigcup Agent.\texttt{parts}(\texttt{initState}\,Agent)$$
$$\texttt{used}((\texttt{Says}\,A\,B\,X)\#evs) \stackrel{\text{def}}{=} \texttt{parts}\{X\} \cup \texttt{used}\,evs$$
$$\texttt{used}((\texttt{Notes}\,A\,X)\#evs) \stackrel{\text{def}}{=} \texttt{parts}\{X\} \cup \texttt{used}\,evs$$

With the given definitions we are able to express the fact that a message, e.g. a key $K \in \texttt{parts}(\texttt{spies}\,evs)$, may appear in the traffic on the trace evs but the attacker may not be able to extract the key $K \notin \texttt{analz}(\texttt{spies}\,evs)$ even the attacker fakes messages $X \in \texttt{synth}(\texttt{analz}(\texttt{spies}\,evs))$. Many relations have been proven for the operators and functions, which can be found in (Pau98). We give them when appropriate. In conclusion we can say the attacker is able to

- read all messages of the entire network (function `spies`),
- send fraudulent messages to honest agents (`Fake-Rule`),
- intercept messages (no firing of rules) and to
- control an arbitrary set of compromised agents (`Oops-Rule`).

In the next section we have a look at an example protocol and examine useful properties to be proven.

8.2 A smart card authentication protocol

In the following we consider a protocol for the mutual authentication between an external application AP, e.g. running on a host computer, and a smart card SC connected to the computer. The protocol additionally establishes a session key for the confidential transmission of data. The challenge-response protocol is a standardized protocol for smart cards described in ISO/IEC 9798-2 and starts with an `askRandom` command in the first step by the application AP, as given in Figure 8.1.

The command `askRandom` is answered by the smart card with a challenge, the nonce N_{sc} in step 2. The application generates its own challenge, the nonce N_{ap}, and encrypts both nonces together with its own identity and the card's identity with the pre-shared authentication key K_{apsc_a} in step 3. The card decrypts the received cipher text and compares the received response with the sent challenge. In the positive case the card generates a new nonce N'_{sc} in step 4, which is not a challenge, and encrypts it together with the challenge N_{ap} and the application's identity with key K_{apsc_a}. The

1. $AP \longrightarrow SC : \ \{\!|\texttt{askRandom}, AP|\!\}$
2. $SC \longrightarrow AP : \ N_{sc}$
3. $AP \longrightarrow SC : \ \{\!|N_{ap}, N_{sc}, SC, AP|\!\}_{K_{apsc_a}}$
4. $SC \longrightarrow AP : \ \{\!|N_{ap}, N'_{sc}, AP|\!\}_{K_{apsc_a}}$
5. $AP \longrightarrow SC : \ \{\!|\texttt{getSessionKey}, N'_{sc}|\!\}_{K_{apsc_e}}$
6. $SC \longrightarrow AP : \ \{\!|N'_{sc}, K_{apsc_s}|\!\}_{K_{apsc_e}}$

Figure 8.1: The mutual authentication protocol

aim of the new nonce N'_{sc} is to make the cipher in step 4 different to the cipher in step 3. The application also checks if both challenges match and in the positive case asks the card for a new session key with the standardized command `getSessionKey` in step 5. The encryption with the pre-shared key K_{apsc_e} of the command together with the session identifier N'_{sc} could be omitted for security reasons but is considered here to bind the command to the particular protocol session. Furthermore, in smart card standards the protocol step 3 and step 4 are combined in the command `ExternalAuthenticate`. Finally, the card sends the generated session key K_{apsc_s} together with N'_{sc} and encrypted with the encryption key K_{apsc_e} to application AP in step 6. The established session key can be used for *secure messaging* in subsequent steps. It ensures confidentiality and integrity of subsequent messages. The formalization and verification of a protocol that goes beyond the session key generation can be found in (CRS+06) in the context of a *Chipcard based Biometric Identification System.*

The mutual authentication protocol shall ensure that no application AP' other than application AP can successfully be authenticated by the card and vice versa. We want to make sure that the new session key K_{apsc_s} is unique for a particular session and is known to both partners only. In order to formally analyze the protocol regarding the properties we need to model the protocol and the attacker in terms of rules of the inductive approach.

8.2.1 Protocol formalization

The inductive model of the mutual authentication protocol consists of ten rules, where six rules belong to the six protocol steps, two rules belong to the attacker and another two rules define traces. The model is given in Figure 8.2 that we describe in the following.

The aim of the model is to describe valid *map* traces out of the infinite set of traces. By definition the empty trace is a *map* trace, $[\,] \in \texttt{map}$, which is modeled in rule `mapNil`. The membership of an extended trace to the set of *map* traces depends on the prior trace and the last event. It is: the extended

mapNil: $[] \in map$

mapSays1: $evs1 \in map \wedge AP \neq SC$
$\Longrightarrow (Says\ AP\ SC\ \{|askRandom, AP|\}) \# evs1 \in map$

mapSays2: $evs2 \in map \wedge SC \neq AP \wedge (Gets\ SC\ \{|askRandom, AP|\}) \in evs2 \wedge$
$N_{sc} \notin used\ evs2$
$\Longrightarrow (Says\ SC\ AP\ N_{sc}) \# evs2 \in map$

mapSays3: $evs3 \in map \wedge AP \neq SC \wedge (Gets\ AP\ N_{sc}) \in evs3 \wedge$
$N_{ap} \notin used\ evs3$
$\Longrightarrow (Says\ AP\ SC\ Crypt\ K_{apsc_a}\ \{|N_{ap}, N_{sc}, SC, AP|\}) \# evs3 \in map$

mapSays4: $evs4 \in map \wedge SC \neq AP \wedge N'_{sc} \notin used\ evs3 \wedge$
$(Gets\ SC\ Crypt\ K_{apsc_a}\ \{|N_{ap}, N_{sc}, SC, AP|\}) \in evs4$
$\Longrightarrow (Says\ SC\ AP\ Crypt\ K_{apsc_a}\ \{|N_{ap}, N'_{sc}, AP|\}) \# evs4 \in map$

mapSays5: $evs5 \in map \wedge AP \neq SC \wedge$
$(Gets\ AP\ Crypt\ K_{apsc_a}\ \{|N_{ap}, N'_{sc}, AP|\}) \in evs5$
$\Longrightarrow (Says\ AP\ SC\ Crypt\ K_{apsc_e}\ \{|getSessionKey, N'_{sc}|\}) \# evs5 \in map$

mapSays6: $evs6 \in map \wedge SC \neq AP \wedge K_{apsc_s} \notin used\ evs6 \wedge$
$(Gets\ SC\ Crypt\ K_{apsc_e}\ \{|getSessionKey, N'_{sc}|\}) \in evs6$
$\Longrightarrow (Says\ SC\ AP\ Crypt\ K_{apsc_e}\ \{|N'_{sc}, K_{apsc_s}|\}) \# evs6 \in map$

mapFake: $evsF \in map \wedge Spy \neq B \wedge X \in synth(analz(spies\ evsF))$
$\Longrightarrow (Says\ Spy\ B\ X) \# evsF \in map$

mapOops: $evsO \in map \wedge (Says\ SC\ AP\ Crypt\ K_{apsc_e}\ \{|N'_{sc}, K_{apsc_s}|\}) \in evsO$
$\Longrightarrow (Notes\ Spy\ \{|N'_{sc}, K_{apsc_s}|\}) \# evsO \in map$

mapRecp: $evsR \in map \wedge (Says\ A\ B\ X) \in evsR$
$\Longrightarrow (Gets\ B\ X) \# evsR \in map$

Figure 8.2: The inductive model of the mutual authentication protocol

trace is a member of the *map* trace, denoted as $ev \# evs \in \texttt{map}$, if the prior trace is already a member $evs \in \texttt{map}$ and the event ev is generated by one of the rules `mapSays1` to `mapSays6`, `mapFake`, `mapOops` or `mapRecp`.

All rules can be applied non-deterministically, which means if two or more rules are activated, i.e. the preconditions hold, there is no strategy which of them fires. Additionally, no rule is forced to fire, which may lead to traces with no activity of one of the agents. In order to keep a protocol running the rule `mapRecp` says whenever a message has been sent it can be received. Formally we say if a trace is a *map* trace $evsR \in \texttt{map}$ and there has been a `Says` event in the past $\texttt{Says}\, A\, B\, X \in evsR$ then the receiving of the message is also a *map* trace $(\texttt{Gets}\, B\, X) \# evsR \in \texttt{map}$. In this way we do not model *fairness* but *reachability*, because a trace that never applies the `mapRecp` rule is a valid *map* trace. Please note, security analysis of cryptographic protocols is not primarily concerned with fairness but with data confidentiality, authenticity and integrity. That is why we assume the rule `mapRecp` to be applied eventually.

The rules `mapSays1` to `mapSays6` represent the protocol step 1 to step 6. We exemplary describe rule `mapSays3`. It states that if a trace $evs3$ is a *map* trace $evs3 \in \texttt{map}$ and there has been a `Gets` event in the past (i.e. AP successfully received the challenge from SC) and AP, SC are not the same agents and AP generated a fresh nonce $N_{ap} \notin \texttt{used}\, evs3$ then the sending of the response is also a *map* trace

$$(\texttt{Says}\ AP\ SC\ \texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap}, N_{sc}, SC, AP|\!\}) \# evs3 \in \texttt{map}.$$

The remaining protocol steps are modeled in the same fashion.

In the inductive model the attacker is defined by the rules `mapFake` and `mapOops`, whereby the former rule allows the creation of new messages taken from the knowledge base of the attacker. Formally we say, if the trace $evsF$ is a *map* trace $evsF \in \texttt{map}$ and the attacker does not send the message to himself $Spy \neq \texttt{B}$ and the message X can be created from the attacker's knowledge base of the $evsF$ trace $X \in \texttt{synth}(\texttt{analz}(\texttt{spies}\, evsF))$ then the attacker may send the message X to an arbitrary agent B.

The latter rule models the 'oops' case that is used to investigate whether revealed session keys can be exploited to attack other protocol sessions. This is formalized with the event $\texttt{Notes}\ Spy\ \{\!|N'_{sc}, K_{apsc_s}|\!\}$ that may expand the *map* trace if the event $\texttt{Says}\ SC\ AP\ \texttt{Crypt}\, K_{apsc_e}\, \{\!|N'_{sc}, K_{apsc_s}|\!\}$ has occurred before in the *map* trace $evsO$.

We already mentioned that the mutual authentication protocol shall ensure authenticity of both partners as well as confidentiality and freshness of the session key in order to use the session key for a confidential communication or for integrity of the transferred data in subsequent steps. In the next section we formalize the objectives as properties of the inductive model.

8.2.2 Security properties

In this section we work out properties to be fulfilled by the mutual authentication protocol. The very first property to be verified regards the proper transcription of protocol steps into rules. The so-called *possibility property* states that there are protocol traces that include the last step of the protocol and hence, shows that the model allows the completion of a protocol run. The proof is done by showing that the preconditions of all rules can be met.

Regularity lemmas are concerned with conditions on the appearance of a message X in the traffic evs of a protocol $X \in \texttt{parts}(\texttt{spies}\, evs)$. For instance, a basic regularity law states that secret keys of an agent A remain secret in an arbitrary protocol trace evs:

$$\texttt{Key}(\texttt{shrK}\, A) \in \texttt{parts}(\texttt{spies}\ evs) \Longleftrightarrow A \in \texttt{bad}.$$

The statement is very strong because it does not allow the encrypted transmission of long-term keys, which might be necessary in a key delivery protocol. In this case the law shall be expressed in terms of the `analz` operator. Because the subset relation $\texttt{analz}\, H \subseteq \texttt{parts}\, H$ holds this may allow keys to be transferred but not to be available to the attacker:

$$\texttt{Key}(\texttt{shrK}\, A) \in \texttt{analz}(\texttt{spies}\ evs) \Longleftrightarrow A \in \texttt{bad}.$$

Those regularity lemmas are needed for proving the authenticity or confidentiality of messages that might be encrypted with long-term keys.

Authenticity theorems are concerned with the truthfullness of origin of a message. The protocol shall provide the mutual authentication between application AP and smart card SC. From the smart card point of view the authentication of AP is performed in step 3 of the protocol. The message of this step contains the challenge N_{sc} sent before to AP and is encrypted with the shared authentication key K_{apsc_a}. *Authenticity* in that context means the smart card SC is convinced that AP is indeed the sender of this message. We formalize the property in the following theorem.

Theorem 8.1 (AP-Auth-Thm)

$$\begin{aligned}
&\forall\, evs,\, SC,\, AP,\, N_{ap},\, N_{sc}: \\
&\quad evs \in \texttt{map} \wedge\ SC \notin \texttt{bad} \ \wedge\ AP \notin \texttt{bad} \\
&\quad \wedge\, \texttt{Says}\, SC\, AP\, N_{sc} \in evs \\
&\quad \wedge\, \texttt{Gets}\, SC\, \texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\, N_{sc},\, SC,\, AP|\!\} \in evs \\
&\Longrightarrow \texttt{Says}\ AP\ SC\ \texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\, N_{sc},\, SC,\, AP|\!\} \in evs
\end{aligned}$$

Theorem 8.1 says from the smart card point of view for an arbitrary *map* trace evs: if the smart card first sent the challenge and second received the response and third the smart card as well as the application are not compromised then the smart card shall infer that the response has indeed sent by the application AP. This statement depends on the authenticity of the transferred response. Thus, in order to prove the theorem we need to proof the following lemma first.

Lemma 8.1 (AP-Auth-Lem)
$\forall\, evs,\ SC,\ AP,\ N_{ap},\ N_{sc}$:
$evs \in \texttt{map} \wedge\ SC \notin \texttt{bad} \wedge\ AP \notin \texttt{bad}$
$\wedge\, \texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\ N_{sc},\ SC,\ AP|\!\} \in \texttt{parts}(\texttt{spies}\, evs)$
$\Longrightarrow \texttt{Says}\ AP\ SC\ \texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\ N_{sc},\ SC,\ AP|\!\} \in evs$

Lemma 8.1 states if a message X appears in the set $\texttt{parts}(\texttt{spies}\ evs)$ then it must have been sent in the traffic of an arbitrary *map* trace evs. The proof of this lemma follows from the regularity lemma $X \in \texttt{parts}(\texttt{spies}\, evs)$ that we discussed above. In order to illustrate the lemma we consider the attacker creating a fraudulent response and hence, faking the application AP. This leads to the following lemma:

Lemma 8.2
$\forall\, evsF \in \texttt{map},\ SC,\ AP,\ N_{ap},\ N_{sc}$:
$\texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\ N_{sc},\ SC,\ AP|\!\} \notin \texttt{parts}(\texttt{spies}\, evsF) \wedge$
$\texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\ N_{sc},\ SC,\ AP|\!\} \in \texttt{synth}(\texttt{analz}(\texttt{spies}\, evsF))$
$\Longrightarrow \texttt{Says}\ AP\ SC\ \texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\ N_{sc},\ SC,\ AP|\!\} \in evsF$

The premises of Lemma 8.2 makes the assumption that the response did not appear on the traffic on $evsF$ but has been synthesized from the analysis of the previous traffic by the attacker. By the definition of operator `synth` there are two possibilities fulfilling the assumption. Firstly, the response is taken on the whole from $\texttt{analz}(\texttt{spies}\, evsF)$, which could be interpreted as simply forwarding the response. Because of $\texttt{analz}\ H \subseteq \texttt{parts}\ H$ it should appear in $\texttt{parts}(\texttt{spies}\, evsF)$ as well, which contradicts the first assumption. Secondly, the attacker created the response out of its components, which means $\{SC,\ AP,\ N_{ap},\ N_{sc},\ K_{apsc_a}\} \in \texttt{analz}(\texttt{spies}\ evsF)$. This is in fact true for the agent name AP and the challenge N_{ap}, since they are transferred in clear in step 1 and step 2. But by the basic regularity lemma the long-term key K_{apsc_a} is in the set only if SC or AP are compromised, e.g. $AP \in \texttt{bad}$. Thus, for authentication statements as in Theorem 8.1 and Lemma 8.1 we have to assume the agents not to be compromised.

We formulate a similar theorem and lemma from the application point of view to authenticate the smart card. Here, Theorem 8.2 relies on Lemma 8.3 that describes the authenticity of the fourth message in the protocol.

Theorem 8.2 (SC-Auth-Thm)
$\forall\, evs,\ SC,\ AP,\ N_{ap},\ N_{sc},\ N'_{sc}$:
$evs \in \texttt{map} \wedge\ SC \notin \texttt{bad} \wedge\ AP \notin \texttt{bad}$
$\wedge\, \texttt{Says}\ AP\ SC\ \texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\ N_{sc},\ SC,\ AP|\!\} \in evs$
$\wedge\, \texttt{Gets}\, AP\, \texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\ N'_{sc},\ AP|\!\} \in evs$
$\Longrightarrow \texttt{Says}\ SC\ AP\ \texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\ N'_{sc},\ AP|\!\} \in evs$

Lemma 8.3 (SC-Auth-Lem)
$\forall\, evs,\, AP,\, SC,\, N_{ap},\, N'_{sc}$:
$evs \in \texttt{map} \wedge\ SC \notin \texttt{bad} \wedge\ AP \notin \texttt{bad}$
$\wedge\, \texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\, N'_{sc},\, AP|\!\} \in \texttt{parts}(\texttt{spies}\, evs)$
$\Longrightarrow \texttt{Says}\ SC\ AP\ \texttt{Crypt}\, K_{apsc_a}\, \{\!|N_{ap},\, N'_{sc},\, AP|\!\} \in evs$

With a proof of the proposed theorems and lemmas we verify the correctness of the protocol regarding the mutual authentication between an application and a smart card. Based on the established trust level both partners shall share a session key in order to communicate confidentially in subsequent steps. We want the session key to be known to both partners only and to be fresh, which leads to new proof obligations.

Confidentiality theorems are concerned with messages that should not be disclosed to the attacker. This is expressed in terms of the set $\texttt{analz}(\texttt{spies}\ evs)$. For protocol we formulate the following theorem.

Theorem 8.3 (Key-Conf-Thm)
$\forall\, evs,\, SC,\, AP,\, K_{apsc_s},\, N'_{sc}$:
$evs \in \texttt{map} \wedge\ SC \notin \texttt{bad} \wedge\ AP \notin \texttt{bad}$
$\wedge\, \texttt{Says}\ SC\ AP\ \texttt{Crypt}\, K_{apsc_e}\, \{\!|\, N'_{sc},\, K_{apsc_s}|\!\} \in evs$
$\wedge\, \texttt{Notes}\, Spy\{\!|N'_{sc},\, K_{apsc_s}|\!\} \notin evs$
$\Longrightarrow K_{apsc_s} \notin \texttt{analz}(\texttt{spies}\, evs)$

Theorem 8.3 expresses the inability of the attacker to obtain the session key K_{apsc_s} within a session of the protocol. We assume for an arbitrary *map* trace $evs \in \texttt{map}$ including the sending of the session key according to step 6 $\texttt{Says}\ SC\ AP\ \texttt{Crypt}\, K_{apsc_e}\, \{\!|\, N'_{sc},\, K_{apsc_s}|\!\} \in evs$ and not including the 'oops' event $\texttt{Notes}\, Spy\{\!|N'_{sc},\, K_{apsc_s}|\!\} \notin evs$ that the attacker can not obtain the session key. Furthermore, we need to show the authenticity of the session key from the application point of view because the session key is generated by the smart card without any input from the application. We express the authenticity of the sent session key in Theorem 8.4.

Theorem 8.4 (Key-Auth-Thm)
$\forall\, evs,\, SC,\, AP,\, N_{ap},\, N'_{sc},\, K_{apsc_s}$:
$evs \in \texttt{map} \wedge\ SC \notin \texttt{bad} \wedge\ AP \notin \texttt{bad}$
$\wedge\, \texttt{Crypt}\, K_{apsc_a}\, \{\!|\, N_{ap},\, N'_{sc},\, AP|\!\} \in \texttt{parts}(\texttt{spies}\, evs)$
$\wedge\, \texttt{Crypt}\, K_{apsc_e}\, \{\!|\, N'_{sc},\, K_{apsc_s}|\!\} \in \texttt{parts}(\texttt{spies}\, evs)$
$\Longrightarrow \texttt{Says}\ SC\ AP\ \texttt{Crypt}\, K_{apsc_e}\, \{\!|\, N'_{sc},\, K_{apsc_s}|\!\} \in evs$

Since the theorem expresses the authenticity of the message it does not state the freshness of the session key, which we formulate in a further theorem.

Unicity theorems are concerned with the creation of fresh components such as nonces and session keys. Because a fresh component shall not appear more than once it is uniquely bound to its message of origin. This leads

to the statement: if two separate messages of separate events contain the same certain component assumed to be fresh then the both events must be identical. For the delivery of the session key in step 6 of the protocol we formulate Theorem 8.5.

Theorem 8.5 (Key-Unicity-Thm)
$\exists\, SC', AP', N''_{sc}.\ \forall\, evs, SC, AP, N'_{sc}:$
$evs \in \texttt{map}$
$\wedge\ K_{apsc_s} \notin \texttt{analz}(\texttt{spies}\, evs)$
$\wedge\ \texttt{Says}\ SC\ AP\ \texttt{Crypt}\, K_{apsc_e}\, \{\!|\, N'_{sc}, K_{apsc_s} |\!\} \in evs$
$\Longrightarrow SC = SC' \wedge AP = AP' \wedge N'_{sc} = N''_{sc}$

The free occurrence of the encrypted session key K_{apsc_s} uniquely determines the other three components. Hence, the theorem expresses the binding of the session key to the protocol run that includes the generation of the nonce N'_{sc}. In this way we exclude the replay of the session key from an earlier protocol run. The proof of the theorem requires applying the authenticity Theorem 8.4 of the message sent by the smart card. Please note, we assume the smart card to always generate fresh session keys. In case a smart card generates a session key twice, e.g. because of a lousy implemented random source, and thus instantiates the session keys K_{apsc_s} and K'_{apsc_s} with the same value this would not be detected by Theorem 8.5. We go into more detail on this topic in Section 9.2.

8.3 Formalization in VSE-II

The theorems given in the previous section describe the desired objectives of the mutual authentication protocol, which become security properties in the formal analysis. The verification of the theorems is done in the interactive theorem prover of VSE-II. To perform the proofs we first have to implement the protocol in VSE-SL. Therefore, VSE-II provides

- a library of predefined VSE theories according to the Paulson approach as basic notions for formalizing protocols including abstract data types for agents, keys the operators `parts`, `analz` and `synth` and functions `spies` and `used` and so forth;
- an extension of the front end of VSE supporting the user friendly specification of individual protocols and the automatic generation of lemma bases;
- a strategy for the interactive generation of inductive proofs about protocols.

In this section we first give the VSE-SL specification of the protocol and the security properties to be proven and then describe a proof sketch.

8.3.1 VSE model of the authentication protocol

For the definition of abstract data types the VSE-SL provides a development object of type `THEORY` as we for instance already used it for the basic definitions of `lists` in the specification of the Simple Smart Card Model. For the specification of protocols we are not concerned with states and state transitions and thus, do not use development objects of type `TLSPEC`. The protocol proofs are inductively performed within the structure of the abstract data types.

We define a new object `THEORY T_MAP` that includes all abstract data types of the Paulson approach summarized in `THEORY TProtocol` in the slot `USING TProtocol`. The Development Graph of the mutual authentication protocol specification is given in Figure 8.3. Objects of type `THEORY` are illustrated as pentagons in the Development Graph. In the next slots we define functions, predicates and variables needed for the specification that we describe in following.

```
THEORY T_MAP
 USING TProtocol
 FUNCTIONS Adm : AgentT;
           askRandom, getSessKey : nat
 PREDICATES MAP : ProtocolTrace;
            MAP_Says1, MAP_Says2, MAP_Says3, MAP_Says4, MAP_Says5,
            MAP_Says6, MAP_Oops : ProtocolEvent, ProtocolTrace
 VARS ev : ProtocolEvent;
      evs : ProtocolTrace;
      SC, AP : AgentT;
      Nsc, Nsc2, Nap : nat;
      Kapsc : KeyT
```

In the slot `FUNCTIONS` we define `Adm` to be a function of type `AgentT` that is itself defined in a basic (freely generated) data type object `BASIC` not illustrated in the Development Graph. The complete protocol library is about 1100 lines of specification code and about 31 nodes in the Development Graph so that we can give a selection only. Development objects of type `BASIC` have a different figure in the graph: a pentagon with a double line on the top and bottom. An agent specified in `AgentT` can be either trusted by default `secureAg(NAT)` or known by other agents `friend(NAT)` or the spy. Each agent is uniquely identified by a natural number. The `WITH` clauses introduce predicates checking for certain data types.

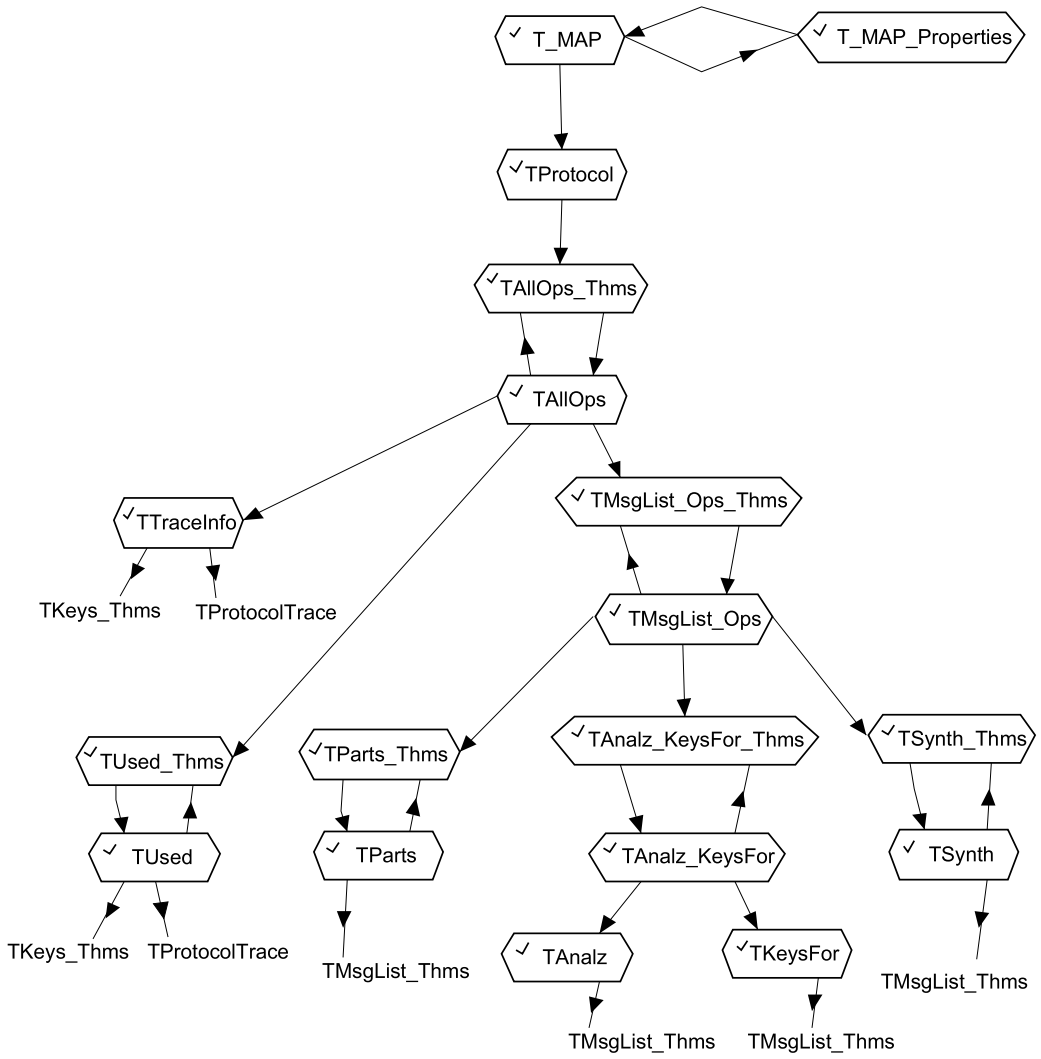


Figure 8.3: VSE Development Graph of the mutual authentication protocol

```
BASIC BAgent
 USING NATURAL
 /* Data type for agents: */
 AgentT = secureAg(secureAgNr : NAT) WITH isSecure |
          friend(friendNr : NAT) WITH isFriend |
          spy WITH isSpy
BASICEND
```

Moreover, we define the functions `askRandom` and `getSessionKey` to specify the standard smart card command in the protocol. Next, we define predicates determining the membership of an arbitrary trace to the set of *map* traces that are refined in the `AXIOMS` slot below. The predicate `MAP` is defined over a protocol trace `ProtocolTrace`, and the predicates `MAP_Says1...MAP_Says6` are defined over `ProtocolEvent` and `ProtocolTrace` indicating the desired extensions of a valid *map* trace according to the protocol rules. A protocol event can be either a `Says`, `Gets` event or `Notes` event. For instance a `Says` event consists of a sender of type `AgentT`, an intended receiver of type `AgentT` and the message `Msg` to be sent.

```
BASIC BProtocolEvent
 USING TMsgList_Thms
 /* Data type for protocol events: */
 ProtocolEvent = Says(sender : AgentT,
                      address : AgentT,
                      sentMsg : Msg) WITH isSays |
                 Gets(receiver : AgentT,gotMsg : Msg) WITH isGets |
                 Notes(subject : AgentT,noteMsg : Msg) WITH isNotes
BASICEND
```

Based on the definition of protocol events a protocol trace is either the empty trace `nullEvent` or the concatenation of a trace with a new event `addEvent`.

```
BASIC BProtocolTrace
 USING BProtocolEvent;
       NATURAL
 /* Data type for protocol traces: */
 ProtocolTrace = nullEvent WITH isNullEvent |
                 addEvent(lastEvent : ProtocolEvent,
                          preEvents : ProtocolTrace) WITH isAddEvent
BASICEND
```

Lastly, in the slot `VARS` all the variables used in the specification of the protocol are defined as there are agents representing the smart card `SC` and the external application `AP` or nonces `Nsc`, `Nap` belonging to `SC` and `AP` or the session key `Kscap`. Next, we have to specify the defined predicates in

the AXIOMS slot. First of all, we say that the empty trace is a valid *map* trace MAP(nullEvent) named MAPNull. An extension of a given *map* trace MAP(evs) with a new event ev results in a valid extended *map* trace if and only if the event is added according to the protocol rules represented by the predicates MAP_Says1(ev,evs)...MAP_Says6(ev,evs), MAP_Oops(ev,evs), Fake_event(ev,evs) and Gets_event(ev,evs).

```
AXIOMS
  secureAg0 : Adm = secureAg(0);
  notEqaskRandom_getSessKey : NOT askRandom = getSessKey;
  MAPNull : MAP(nullEvent);
  MAPAdd : MAP(addEvent(ev,evs)) <->
           (MAP(evs) AND
            (MAP_Says1(ev,evs) OR MAP_Says2(ev,evs) OR
             MAP_Says3(ev,evs) OR MAP_Says4(ev,evs) OR
             MAP_Says5(ev,evs) OR MAP_Says6(ev,evs) OR
             MAP_Oops(ev,evs) OR
             Fake_event(ev,evs) OR
             Gets_event(ev,evs)));
```

All the predicates represent the protocol rules given in Figure 8.1 on page 129. We exemplary discuss the specification of step 3 of the protocol given in the inductive model as follows.

mapSays3: $evs3 \in map \wedge AP \neq SC \wedge (Gets\ AP\ N_{sc}) \in evs3 \wedge N_{ap} \notin used\ evs3 \Longrightarrow (Says\ AP\ SC\ Crypt\ K_{apsc_a}\ \{\!|N_{ap}, N_{sc}, SC, AP|\!)\ \#\ evs3 \in map$

In VSE-SL we specify the predicate MAP_Says3(ev,evs) to be true for a given event ev and trace evs if and only if first, there exist agents AP and SC and nonces Nap and SC with nonce Nap has not been used so far in the particular trace evs and thus, is not element of the set used(evs). Second, The rules mapSays1 and mapSays2 have been applied meaning messages askRandom and the random itself represented by Nsc have been sent from the perspective of the agent AP. Third, the new event ev is the Says event representing the sending of the cipher holding the responds and the new challenge for the smart card.

```
  MAPSays3 : MAP_Says3(ev,evs) <->
             EX AP, SC, Nsc, Nap :
             (NOT msgIN(nonce(Nap),used(evs)) AND
              eventIN(Says(AP,SC,pair(num(askRandom),
                           agent(AP))),evs) AND
              eventIN(Gets(AP,nonce(Nsc)),evs) AND
              ev = Says(AP,SC,crypt(msk(SC,AP,0),
```

```
pair(nonce(Nap),pair(nonce(Nsc),
pair(agent(SC),agent(AP))))));
```

The preconditions of rule `mapSays3` saying that sender and receiver shall not be identical and that trace `evs` must already be a valid *map* trace is formalized with the bounding of `AP` and `SC` and with the condition `MAP(evs)` in the `MAPAdd` predicate. Hence, the complete rule `mapSays3` is specified with the axiom:

```
MAP(addEvent(ev,evs)) <-> MAP(evs) AND MAP_Says3(ev,evs).
```

The specification of the entire protocol including the `Oops-Rule` is given in Appendix B.1. Because the `Fake-Rule` and the `mapRecp` representing the `Gets` rule is identical to all protocols we do not specify it individually for the mutual authentication protocol. The rules are given in the basic formalism of `THEORY TProtocol`. Finally, in the slot `SATISFIES` of `THEORY T_MAP` we specify that the formalized protocol shall satisfy the security properties formalized in `T_MAP_Properties`, which is discussed in the next section.

8.3.2 Verification of security properties

We already discussed the desired objectives to be fulfilled by the protocol and formulated the corresponding theorems and lemmas in terms of the inductive model in Section 8.2.2. Proving a property φ of the protocol model given by the protocol rules is performed according to the induction principle over protocol traces *evs*. Therefore, it must be shown that φ holds for the empty trace $\varphi([\,])$ (the base case) and it must be proven an assertion of the form $\varphi(evs) \Rightarrow \varphi(ev\#evs)$ (the induction step) for all possible extensions of trace *evs* according to the protocol rules including the attacker rules `fake` and `oops`.

The theorems have to be specified in VSE-SL as properties of the inductive protocol model, which makes the definition of a new development object `THEORY T_MAP_Properties` necessary. It includes all definitions and data types of the protocol given in `THEORY T_MAP` and additionally holds the specified theorems and lemmas. In the following we discuss a regularity lemma and the confidentiality theorem of the session key exemplary. The complete specification of all properties is given in Appendix B.2.

A basic regularity lemma states that messages, like a secret key `shrK` A of an agent A, remain secret in an arbitrary protocol trace *evs* as long as A is not compromised:

$$\texttt{Key}(\texttt{shrK}\,A) \in \texttt{analz}(\texttt{spies}\ evs) \Longleftrightarrow A \in \texttt{bad}.$$

One may also express the lemma in terms of the operator `parts`, which is a stronger expression because it does not even allow the sending of the key in

encrypted form. Moreover, in the case of shared keys one has additionally to consider all other agents sharing the particular key.

In our specification we customize the regularity lemma by saying that a symmetric key known to agents `A` and `B` can only occur in the knowledge base of the spy if and only if one of the agents is compromised. We will use this lemma in the proof of the confidentiality lemma of the session key K_{apsc_s} between agents SC and AP that SC sends to AP in step 6 of the mutual authentication protocol.

```
THEORY T_MAP_Properties
 USING T_MAP
 AXIOMS    mskRegularityAnalz :
           ALL evs, n, B, A :
           (MAP(evs) ->
           (msgIN(key(msk(A,B,n)),analz(gotInfo(spy,evs))) <->
           (isBad(A) OR isBad(B))));
```

In the above mentioned specification of Lemma `mskRegularityAnalz` the function `msk: AgenT, AgenT, NAT -> keyT` denotes the n-th shared symmetric key between two agents. The predicate `msgIN: Msg, MsgList` yields true if a message `Msg` is contained in a message list `MsgList` and the data type `key` makes a key to a message of type `Msg`. The functions `analz: MsgList -> MsgList` and `gotInfo: AgentT, ProtocolTrace -> MsgList` represent the knowledge of the attacker and predicate `isBad: AgenT` identifies the compromised agents.

For the proof of the authenticity lemmas we will use a stronger regularity lemma because we want the pre-shared authentication and encryption keys K_{apsc_a} and K_{apsc_e} not to be transferred in the protocol. In our scenario we assume the pre-shared keys to be set up in a different procedure, e.g. using a certain key delivery protocol. Hence, we formulate a further regularity lemma in terms of the operator `parts`:

```
 AXIOMS    mskRegularityParts :
           (MAP(evs) AND
            msgIN(key(msk(A,B,n)),parts(gotInfo(spy,evs)))) ->
           (isBad(A) OR isBad(B));
```

Next, we have a look at the Confidentiality Theorem 8.3. The specification in VSE-SL is a direct translation of the theorem. It states that a certain trace *evs* must be a valid *map* trace specified with `MAP(evs)` and must include the event of sending the session key: `Says` $SC\ Ap\ Crypt\ K_{apsc_e}\ \{\!|\ N'_{sc},\ K_{apsc_s}\ |\!\}$, whereby SC and AP shall not be compromised: `NOT isBad(SC)` and `NOT isBad(AP)`. The predicate `eventIN: ProtocolEvent, ProtocolTrace` specifies the membership of an event in a trace. We additionally exclude the 'oops' case `Notes` $Spy\{\!|N'_{sc},\ K_{apsc_s}\ |\!\}$, where the attacker accidentally obtains the session key and can finally conclude that the session key does not occur in the

knowledge base of the attacker. The quantification of the variables `evs`, `SC`, `AP`, `Kapsc` and `Nsc2` is done by the prover when performing the inductive proof. The variable `Nsc2` represents the nonce N'_{sc}.

```
Key_Conf_Thm :
(MAP(evs) AND
 eventIN(Says(SC,AP,crypt(msk(SC,AP,0),
              pair(nonce(Nsc2),key(Kapsc)))),evs) AND
 NOT isBad(SC) AND NOT isBad(AP) AND
 NOT eventIN(Notes(spy,pair(nonce(Nsc2),key(Kapsc))),evs)) ->
 NOT msgIN(key(Kapsc),analz(gotInfo(spy,evs)));
```

In VSE-II all inductive proofs of the protocol properties follow the same structure of proof tasks. Generally, the induction step is proved by transpositioning the premise.

1. Determine the base case and the induction step for the trace structure.
2. Handle the base case.
3. Handle the step case:
   (a) Reduce certain formula to negative assumptions in the premise.
   (b) Add information of the individual protocol steps.
   (c) Reduce the remaining differences and apply the premise.

To perform the proof steps VSE-II offers various *heuristics* in order to close a certain kind of proof goals. For instance, the heuristic `Induction (Protocol Trace)` initializes the inductive proof by choosing the induction variable and reducing the proof goals of the base case and the step case to a simplified normal form. For the base case a heuristic `nullevent` closes the goal by contradiction using axioms about the empty traces like $\forall ev : ev \notin [\,]$. For the handling of the step case the heuristic `Distinction (Protocol Steps)` performs the case distinction over all defined protocol rules and inserts the conditions of the corresponding protocol step.

Moreover, there are various heuristics performing difference reductions between the goal assumptions and corresponding subformulas of the implication premises. As an example we consider the proof attempt of Theorem 8.3 stating the confidentiality of the transferred session key K_{apsc_s} from the smart card to the external application. The arising difference marked in bold face between `Notes` $Spy\{\!|N'_{sc}, K_{apsc_s}|\!\} \notin evs$ in the premises and `Notes` $Spy\{\!|N'_{sc}, K_{apsc_s}|\!\} \notin \mathbf{ev}\#evs$ for a certain new event ev of a particular protocol step is eliminated by these heuristics. Figure 8.4 shows the proof

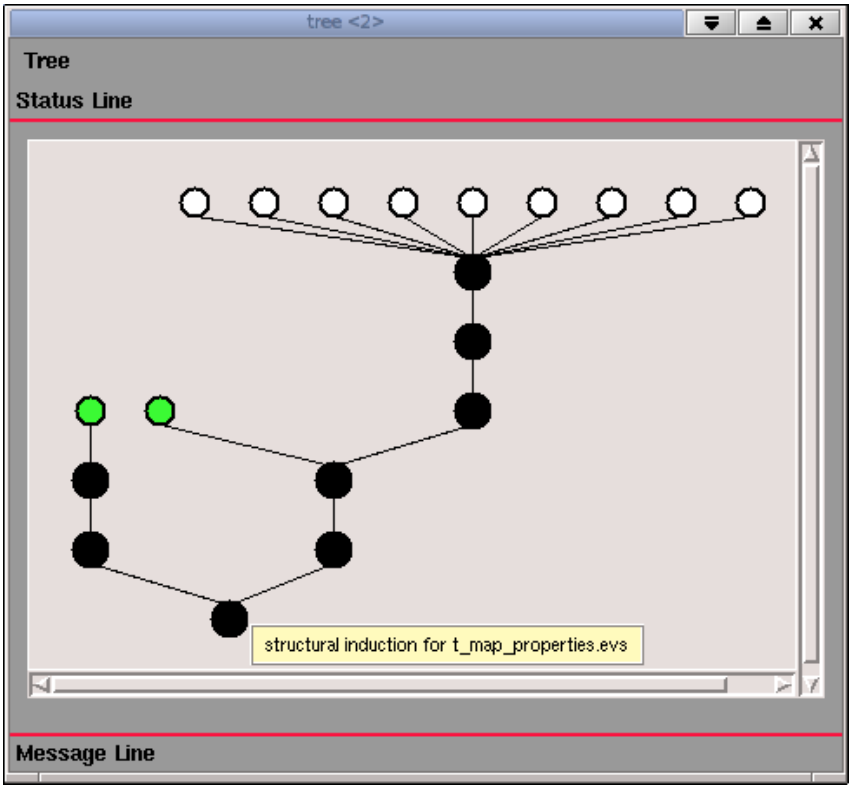


Figure 8.4: VSE proof tree window of the Confidentiality Theorem

tree of Theorem 8.3 after applying the induction heuristic, closing the proof goal of the base case and applying the case distinction rule. The white circles denote open goals holding subgoals for the events `mapSays1`...`mapSays6`, `mapFake`, `mapOops` and `mapRecp`. The complete proof consists of 710 proof steps and 64 interactions, which leads a degree of automation of 90,2%. All theorems and lemmas, except the regularity lemma `mskRegularityAnalz`, have been proven in this fashion.

8.4 Conclusion

The aim of this chapter is the specification of a cryptographic protocol refining the transformation function T as discussed in Chapter 7 in the context of the Simple Smart Card Model (SSCM). Therefore, we introduce a protocol providing both a mutual authentication between two participants and the establishment of a session key, which can be used for securing a subsequent communication between the two partners. The challenge-response protocol assumes pre-shared secret keys K_{apsc_a} and K_{apsc_e} for authentication and en-

cryption purposes to be known to a certain smart card SC and a certain external application AP only. After the successful mutual authentication the smart card generates a new session key and sends it in encrypted form using K_{apsc_e} to the external application.

The transformation function T together with the authentication token a and the validation function v as given in Section 7.2.1 define a secure channel between the smart card represented by the communication component `CommC` and the external application `ExtAppl` in the SSCM. In the mutual authentication protocol the agent SC represents component $CommC$, and agent AP represents component $ExtAppl$. In this regard the message $\{\!|N_{ap}, N_{sc}, SC, AP|\!\}_{K_{apsc_a}}$ sent in step 3 of the protocol instantiates the authentication token a of a particular external application. The validity of a can be checked by component `CommC` via the validation function v instantiated with the decryption function using the authentication key K_{apsc_a} and the matching of the sent challenge N_{sc} with the received responds N_{sc}. The encrypted sending of the session key in step 6 of the protocol $\{\!|N'_{sc}, K_{apsc_s}|\!\}_{K_{apsc_e}}$ instantiates the transformation function T because the additional nonce N'_{sc} uniquely links the message to the previous authentication between SC and AP. Because the encryption key is only known to SC and AP the retransformation function T^* can be applied only by the correct external application SC.

To stress the correctness of the mutual authentication protocol we formally verify that the session key is freshly generated and known to the both partners only in the presence of an attacker. The attacker is allowed to monitor all the network traffic, to delete sent messages, to create new messages out of his collected knowledge and to send them to all participants. The formal verification of the protocol is done with the inductive approach by Paulson in VSE-II that provides all the necessary constructs of the inductive model.

The proofs show that an instantiation of the transformation function T indeed exists and that the protocol fulfills the transformation's requirements under a given attacker model. In this manner we verified security properties for the external communication that we marked in blue colored boxes in Figure 6.4. Hence, it remains to show that the communication component `CommC` refines the transformation function correctly in accordance to the specified protocol rules. In other words, we have to formally verify that component `CommC` behaves exactly as the agent SC in the protocol specification, which is the topic of the following chapter.

Chapter 9

Linking safety and security

Chapter 6 considers the relations between security threats, security objectives and security functions in order to develop dependable security systems. The analysis of the so-called Security Triangle is central for the quality of a security policy and ultimately of the security system. Formal methods can help to analyze the relations, whereby two different main fields can be identified: formal models of security policy and formal verification of cryptographic protocols. Since in modern security engineering both need to be considered, we are going to combine them in a common verification methodology.

In this chapter we first informally give an idea how behaviors of a TLA system specification model representing the security policy can be used to be compared with Paulson's inductive defined protocol traces. In this way we show that a given system specification model behaves exactly as a particular agent of a given cryptographic protocol, which gives evidence of a correctly implemented verified cryptographic protocol.

Then we work out the formal verification tasks to be done in VSE-II by combining the verified mutual authentication protocol given in Section 8.3 with the verified Simple Smart Card Model discussed in Section 7.3. We therefore apply an observer methodology introduced in (Roc04). Since all specifications and verification tasks are performed in VSE-II, the formal model of the security policy also serves as the abstract system specification and has to be refined in further development steps. In this way we define a framework for the verification of security and safety properties in VSE.

9.1 The basic idea

In Sections 7.3 and 8.3 we separately verified a security policy model and a cryptographic protocol, which made various assumptions on the used mech-

anisms in both models necessary.

In the analysis of the mutual authentication protocol we assume the cryptographic primitives to be secure. The formal verification of the protocol using the inductive approach by Paulson states that a particular session key K_i belongs to exactly one protocol session and thus, can not be used in a parallel or a further protocol session. Because we reason about syntactical symbols K_i and K_j representing keys that are different in the inductive model, the possible instantiation of K_i and K_j with the same value can not be recognized in the model. Semantically, we say the same key is delivered twice in two different variables; for $K_i \neq K_j$ it is $\alpha(K_i) = \alpha(K_j)$ for a given interpretation α of the symbols.

On the other hand, in the formal specification of the Simple Smart Card Model (SSCM) using the TLA we assume various mechanisms, like the validation function v and transformations T and $T^\star$, in order to model a secure channel between the components `CommC` and `ExtAppl`. We show that no key data are generated and delivered twice as well as that the transformation T is always applied on delivered key data. This is expressed in terms of the external application: in the SSCM it holds $\alpha(K_i) \neq \alpha(K_j)$ for all $K_i, K_j \in$ `h3_appl` for a given interpretation α, where the set `h3_appl` holds all delivered key data K_i.

In this section we informally propose an approach that combines verification tasks of security policy models in the TLA and of inductive models of cryptographic protocols in order to discharge the assumptions made in both models. We consider Paulson's inductive approach of verifying cryptographic protocols (Pau98) and Lamport's TLA for specifying and verifying systems (Lam94). The main idea is to compare protocol traces with behaviors of the system specification. Therefore, we briefly summarize Paulson's approach (see Section 8.1) and TLA specifications (see Section 7.1) in order to investigate the resulting verification tasks.

Paulson's inductive approach

The approach by Paulson inductively defines protocols as sets of *traces*, whereas a trace is a list of communication *events*, e.g. sending or receiving a message. The inductive definition of traces lists the possible events that an agent as well as the attacker can perform. Hence, a particular trace is one possible sequence of events. The verification task of a given protocol is to prove that for every state in every trace no security condition fails. This is done by induction on the length of traces.

To illustrate a trace in the Paulson model we consider the challenge-response protocol for mutual authentication between a smart card SC and

an external application AP discussed in Section 8.2. The protocol defines six steps, whereby we distinguish a *send* step and a *gets* step. The corresponding *gets* step of a *send* step is denoted with an additional star * in Figure 9.1. The protocol is modeled in 12 steps; application AP sends an `AskRandom` command in `step` 1 that might be received by the smart card in `step` $1^\star$. Then the smart card SC sends the challenge N_{sc} to AP in `step` 2, which again might be received by application AP in `step` $2^\star$. The application encrypts the challenge N_{sc} together with its own challenge N_{ap} and the identities SC and AP and sends it back to the smart card in `step` 3 that might receive it in `step` $3^\star$. The description of the protocol proceeds in the ping-pong fashion according to the protocol rules given in Figure 8.2.

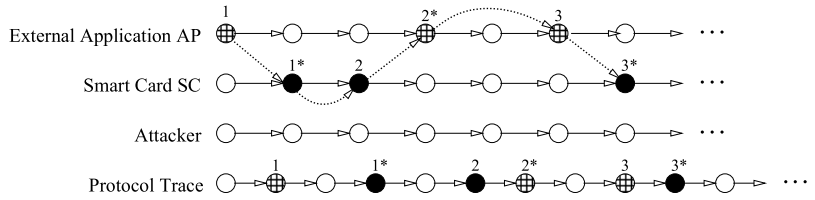


Figure 9.1: Possible traces of events in protocol verification

In the Paulson model every agent together with the attacker defines its own trace of events, which gives a local view on the protocol. All individual traces result in the combined Protocol Trace that represents a global view on the protocol. Examples of possible traces of the given protocol are illustrated in Figure 9.1, whereas events of the smart card SC and the external application AP regarding the given protocol are denoted with patterned circles ⊕ and black circles ●, respectively. The white circles ○ illustrate other events, e.g. by the attacker. For instance, the external application AP may start a protocol session with step 1 and may then perform any other events, e.g. start a new protocol session with a smart card $\widetilde{SC}$. Furthermore, the smart card SC may receive faked messages from the attacker and eventually receives the message in `step` $1^\star$ from the application AP. Moreover, sometimes AP receives the message from the smart card SC in `step` $2^\star$ and may not immediately respond to it but may generate other events and eventually performs `step` 3. In the figure the dotted line indicates the intended protocol run, which becomes the verification task in the protocol analysis on the overall protocol trace. A comprehensive verification of the extended cryptographic protocol in the context of a biometric smart card with the help of VSE-II can be found in (CRS$^+$06).

System specification in TLA

In this section we again consider the Simple Smart Card Model (SSCM) given in Section 7.2.1. The SSCM is specified in temporal logic as a state-based system and assumes the smart card operating system to be decomposable into several components. The state space of a component is divided into *input lines* holding values from the environment, *output lines* used to deliver values back to the environment and *internal variables* that can be accessed by the component itself only. The intended *behavior* of the component is specified by the *initial state* and by possible steps called *actions*. In TLA action definitions have a particular structure to allow *stuttering* steps, which mention those variables of an action that are guaranteed not to change if a stuttering step occurs. A stuttering step represents a step taken by the environment of the component.

The initial state together with the permitted steps describe safety aspects of the component, which can be used to deduce that the system component will never enter undesirable states. This way, modular reasoning within the components helps to analyze the entire system that takes non-deterministic behaviors into account. The SSCM is an example of a composed reactive system specification based on system components and input and output lines as illustrated in Figure 9.2.

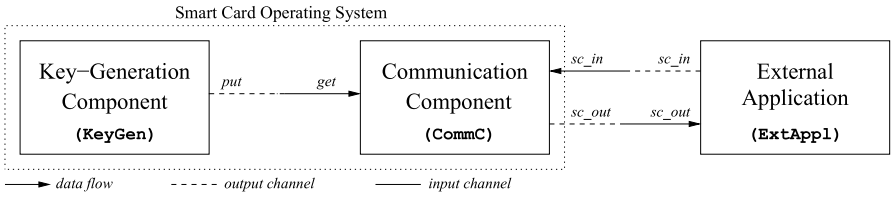


Figure 9.2: A simple smart card system model

In the TLA system specification model we make use of *history variables* that correspond to other variables, e.g. input or output variables, whose histories they represent. The presence of history variables enables us to express conditions in terms of past values of certain state variables. In this model a *trace* is a list of *histories*, whereas a history is the value of the corresponding variable in a specific state. A history also denotes an *event*, e.g. the sending (output) or receiving (input) of values (messages). In this sense each component defines its own trace.

The particular definition of history variables in a specific system specification determines the behaviors we want to look at. In (AL91) history variables together with *prophecy variables* are used to verify the refinement mapping between two abstraction levels of non-deterministic state-machines,

which is needed in the stepwise refinement of the system specification. In (RSW+99; LARS07) history variables provide good means to prove safety and liveness properties in the system specification. Because the history variables for proving safety properties and the history variables for refinement proofs serve for different purposes, they are not necessarily identical in a certain system specification.

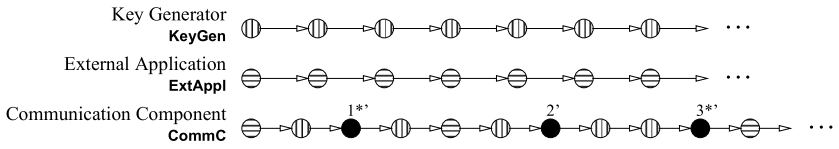


Figure 9.3: Traces of histories in the TLA specification

In Figure 9.3 possible traces of the SSCM are illustrated; the patterned circles ◍ and ⊜ denote input and output events of the components `KeyGen` and `ExtAppl`, respectively. Here we do not distinguish between input and output events. The third trace illustrates events of the communication component `CommC`, where we mark events coming from component `KeyGen` with ◍ circles and events coming from an arbitrary external application `ExtAppl` with ⊜ circles. Additionally, we especially mark events belonging to the cryptographic protocol with black circles ●. These circles denote events where the communication component `CommC` acts on behalf of the smart card SC in the mutual authentication protocol.

Linking protocol traces and history traces

As already mentioned, the modeling and the verification of the cryptographic protocol as well as the system specification is done in the interactive theorem prover VSE-II. The tool support is of special advantage because the definitions of basic abstract data types are used in both the protocol model and the system specification model. With the proper definition of events and histories in the system specification model we are able to compare the Protocol Trace of the protocol model with the History Trace of the Simple Smart Card Model. The idea is to show that the system specification becomes an agent (the smart card obviously) in the protocol model and behaves as expected by the protocol.

In Figure 9.4 we assume the communication component `CommC` of the SSCM given in Figure 9.3 to act on behalf of the smart card SC in Figure 9.1. It is crucial for the comparison of both traces to generate a proper history trace in the system specification model. In the given example of the SSCM it would be sufficient to generate a history trace over the external input and

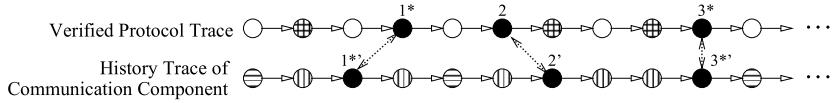


Figure 9.4: Verification task on traces

output events regarding the protocol events only. Then, it remains to show that the reduced history trace of the communication component including steps $1^{\star\prime}, 2', 3^{\star\prime}$ is always a *subset* of the inductively defined (and verified) Protocol Trace including the steps $1^{\star}, 2, 3^{\star}$.

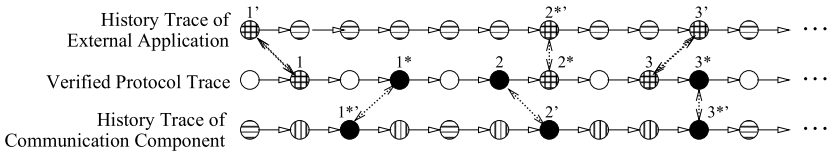


Figure 9.5: Extended verification task on traces

We implicitly assume the other agents of the protocol, like application *AP*, to act as defined in the protocol model. To verify the entire system the external application `ExtAppl` shall be verified in the same fashion as illustrated in Figure 9.5.

9.2 Observer methodology

As pointed out in the previous section our aim is finding a kind of *relation* in order to compare protocol traces of the inductive model with history traces of the TLA model. We define such a relation by means of *observer models* (Roc04) that are concerned with different views on a system. Generally, each view may use a different formalism to specify and analyze the system. For instance, in our scenario we use the TLA formalism to concurrently specify system components together with their communication channels and to analyze information flow between the components of the system. Furthermore, we use the inductive approach by Paulson in order to analyze a cryptographic protocol including steps of an attacker. The protocol is intended to secure the communication channel.

Both models are translated into the specification language VSE-SL and the individual proofs are performed in the theorem prover of VSE-II by showing that a system model *satisfies* a property model. By the construction of VSE-II it holds that the `satisfy` relation (meaning model inclusion) between a VSE-SL representation `VSESL-Model-Spec` of a certain formalism

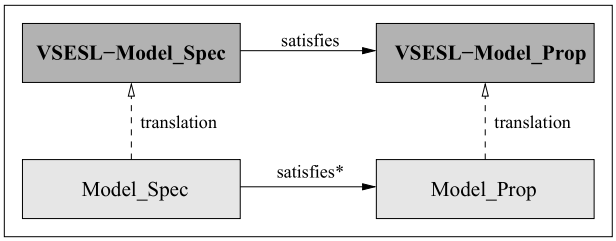


Figure 9.6: Relations between Model Specification and VSE-SL Specification

`Model-Spec` and the corresponding properties `VSESL-Model-Prop` holds if and only if the `satisfy*` relation holds between the ordinary model `Model-Spec` and the corresponding properties `Model-Prop` as illustrated in Figure 9.6. This is provided by VSE-II for the TLA formalism and the inductive protocol model. Moreover, in (Roc04) the VSE-SL specification of Hybrid Automata for the analysis of real-time constraints is introduced but not considered here.

The basic idea of observer methodology is to link the various VSE-SL models by an *observer mapping* realizing a `satisfy` relation. In this way we want to show that a specific system model `Model-Spec` additionally *fulfills* another system model `Model-SpecX` given in a different formalism as shown in Figure 9.7.

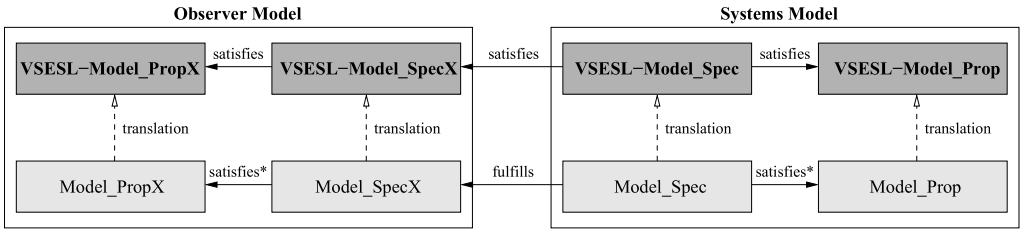


Figure 9.7: Linking VSE-SL Specifications

In our scenario of the Simple Smart Card Model we already specified the SSCM in the TLA formalism and translated it into VSE-SL and verified safety properties. This is illustrated in Figure 9.8 in the red colored box for a general VSE-SL System Model Γ and VSE-SL safety properties ϕ in the context of a security policy model. We separately specified the mutual authentication protocol as an inductive model in VSE-SL and verified security properties denoted as Model $\Gamma^\star$ and VSE-SL properties $\phi^\star$ in the blue colored box in the figure. Here, $\Gamma^\star$ is interpreted as an observer model that has to be linked via an observer mapping. How such mappings look like is discussed in the next section by considering the VSE-SL models of the SSCM and the

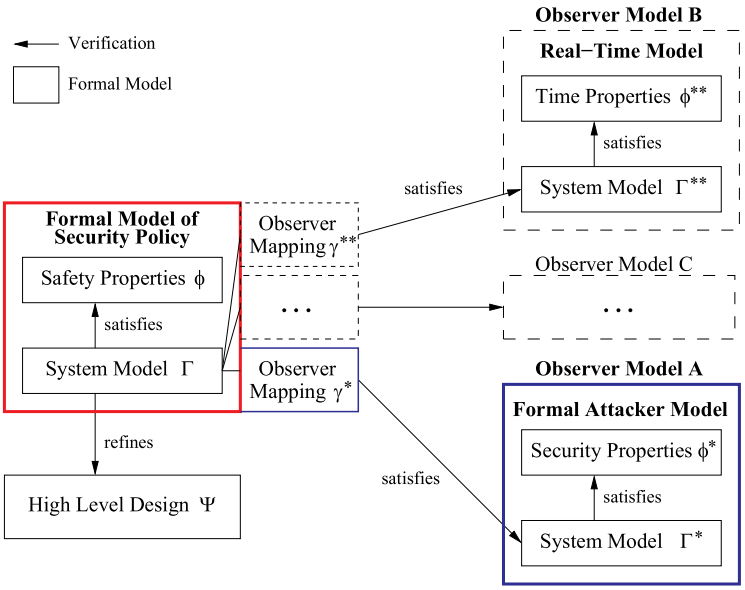


Figure 9.8: Linking VSE-SL Models by means of observer mappings

mutual authentication protocol.

The figure additionally shows further possible observer models as there might be time constraints. As an example one may think of a requirement stating that a session key shall be delivered within a specific time, which would make the inclusion of a real-time model necessary. In this work we are concerned with the red and blue colored boxes only. The red colored box represents the verification of access control mechanisms of the smart card operating system for the internal communication between on-card applications, whereas the blue box represents the verification of security properties of cryptographic protocols used for securing the communication between on-card applications and external applications and devices. We comprehensively discuss this topic in Section 6.2.

In a sense one could even interpret the TLA model of the SSCM as an observer model taking concurrency aspects into account, for instance if the system specification has been modeled in a formalism without temporal constructs. In our scenario we already modeled the system in TLA so that this model becomes the basic model to be refined in further development steps. The question remains, how the inductive protocol model is integrated in the state based system model using an observer mapping, which is the topic of the next sections.

9.3 Extension of the smart card model

The realization of a secure channel in the Simple Smart Card Model between the Communication component `CommC` and the External Application component `ExtAppl` is done with transformation functions T and $T^{\star}$, the authentication token a and validation function v as discussed in Section 8.4. In this section we replace all the mechanisms by steps to be taken by the particular agents of the mutual authentication protocol. Because we are mainly interested in the development of the smart card we concentrate on the component `CommC`.

Performing a certain step in the protocol depends on previous protocol steps, which is formalized in the different protocol rules given in Figure 8.2 on page 130. Component `CommC` needs to remember the steps already taken by itself and by the communication partner, the component `ExtAppl`. Therefore, we consider a set `Stateentrylist` for each communication partner holding information about past events named `StateLabel` and key data shared with trusted communication partners named `StoreItem`. The communication partner represent agents in the inductive protocol model.

$$\texttt{Stateentrylist}_{\texttt{Agent}} \stackrel{\text{def}}{=} \{\, l, i \mid l \in \texttt{StateLabel}, i \in \texttt{StorInfo}\}$$

The set `StateLabel` consists of elements holding information of each single step of the inductive protocol model.

$$\begin{aligned}\texttt{StateLabel} \stackrel{\text{def}}{=} \{&\texttt{start}, \texttt{ready},\\ &\texttt{sent1}(\texttt{Agent}, \texttt{Message}), \texttt{rcvd1}(\texttt{Agent}, \texttt{Message}),\\ &\quad\ldots\\ &\texttt{sent6}(\texttt{Agent}, \texttt{Message}), \texttt{rcvd6}(\texttt{Agent}, \texttt{Message})\}\end{aligned}$$

The entries `start` and `ready` signal that a new protocol run could be started. As we will see later on, `ready` is used by component `CommC` to represent the idle state, where the smart card is waiting for key data requests. The entries `sent1` and `rcvd1` represent the sending and receiving of the message between two agents in step 1 of the protocol. The data types `Agent` and `Messages` are taken from the definitions of the inductive model and represent protocol agents and sent messages. In the same fashion the set `StoreItem` holds key data that belong to a certain communication partner, where we distinguish authentication keys and encryption keys.

$$\begin{aligned}\texttt{StateInfo} \stackrel{\text{def}}{=} \{&\texttt{encKey}(\texttt{Agent}, \texttt{Key}),\\ &\texttt{authKey}(\texttt{Agent}, \texttt{Key})\}\end{aligned}$$

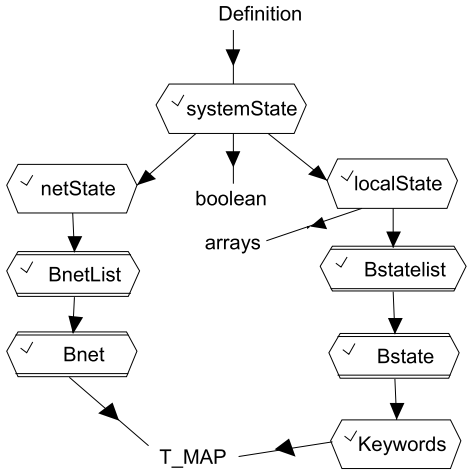


Figure 9.9: VSE Development Graph of system states

Moreover, we define functions over the set regarding the adding and deleting of entries as well as the checking for existence of a certain entry, where a `Stateentry` denotes a `StateLabel` or a `StateInfo`.

$$\texttt{enterstateentry} : \texttt{Agent} \times \texttt{Stateentry} \times \texttt{Stateentrylist} \rightarrow \texttt{Stateentrylist}$$
$$\texttt{delstateentry} : \texttt{Agent} \times \texttt{Stateentry} \times \texttt{Stateentrylist} \rightarrow \texttt{Stateentrylist}$$
$$\texttt{inState} : \texttt{Stateentry} \times \texttt{Stateentrylist} \rightarrow \texttt{Bool}$$

In the VSE-SL model of the *Extended Simple Smart Card Model (eSSCM)* we define a new abstract data type `localState` representing the mentioned set. The local state is defined as a list `stateentrylist` holding entries `stateentry`. The definitions are done in the development objects `BASIC Bstate` and `BASIC Bstatelist`. The complete specification is given in Appendix C.1 and the corresponding Development Graph is illustrated in Figure 9.9.

```
BASIC Bstate
 USING Keywords
 stateentry = stateInfo(stateInfo_arg : StateLabel)
              WITH isstateInfoentry |
              storeInfo(storeInfo_arg : StoreItem)
              WITH isstoreInfoentry

BASIC Bstatelist
 USING Bstate
 stateentrylist = nullentry WITH isnullentry |
                  addstateentry(firststateentry : stateentry,
                                reststateentry : stateentrylist)
                  WITH isaddstateentry
```

In addition to the set `Stateentrylist` representing the local state of a protocol run for a particular agent we define a set representing the global state of all protocol runs, which we will use to perform certain proofs. We define a set `Netentrylist` holding events of sent messages, where an event is a triple consisting of a sender, a receiver and a sent message.

$$\texttt{Netentrylist} \stackrel{\text{def}}{=} \{(s, r, m) \mid s, r \in \texttt{Agent}, m \in \texttt{Messages}\}$$

The definitions are done in the development objects `BASIC Bnet` and `BASIC BnetList`. The type definitions of `Agent` and `Messages` are again imported from the inductive model, which is stated in the `USING` slot of `BASIC Bnet`.

```
BASIC Bnet
 USING T_MAP
 netentry = mknetentry(sender : nat,receiver : nat,entry : Msg)
            WITH isnetentry

BASIC BnetList
 USING Bnet
 Netentrylist = NullNetentry WITH isNullNetentry |
                addnetentry(firstnetentry : netentry,
                            restnetentry : Netentrylist)
                WITH isAddNetEntry
```

As shown in Figure 9.9 all development objects include the definitions of the inductive model of the mutual authentication protocol and are themselves included by development object `Definition` of the eSSCM. The overall Development Graph of the eSSCM is shown in Figure 9.10 and discussed later on. The complete VSE-SL specification is given in Appendix C. Next, we have to enhance the Communication component `CommC` and the External Application component `ExtAppl` with the individual protocol steps.

9.3.1 Cryptographic protocol integration

We use both new data types to model the internal state of a component and the global state according to a protocol run by defining a new variable `OUT StateSC: array` in the VSE-SL specification of the component `CommC` that is available to the combined component of the model. The variable is of type `array` and models the `Stateentrylist` of all instantiations of component `CommC`. The new variable `SHARED INOUT net: Netentrylist` can be accessed by the environment and defines a new history variable for messages sent or received via channels `sc_out` and `sc_in`. Please note, the variables `StateSC` and `net` represent the local and the global view of protocol runs. The variable

StateSC (more precisely one element of the array) will be implemented by the system, whereas net is an auxiliary variable introduced for verification purposes only.

We already used the shared variable key_list in the SSCM that models the key buffer and holds key data generated by component KeyGen. In the extended SSCM we have to additionally handle nonces to be used in the protocol. Because keys are also uniquely generated random numbers with additional characteristics we take key data from the list key_list and use it as nonces. This is due to simplicity in our model. In a further refinement of the system both data generations should be distinguished, because creating key data takes more processing power than creating nonces. The meaning of the new variable SHARED INOUT next: Component is given in the next section. The new definitions of the extended specification of the Communication component TLSPEC CommC are given as follows.

```
TLSPEC CommC
 USING Definition
 DATA
   SHARED INOUT key_list: list    /*channel get from buffer keys*/
   SHARED INOUT next: Component   /*schedule variable*/
   SHARED INOUT net: Netentrylist /*global variable for net state*/
   IN sc_in: Msg                  /*channel sc_in from ExtAppl */
   OUT sc_out: Msg                /*channel sc_out to ExtAppl */
   OUT h2_scout: list             /*history variable of sc_out */
   StateSC: array;                /*array of all local traces */
```

Next, we have to specify the possible actions that can be taken by component CommC. In the SSCM we defined one single action checked_transform() that we want to replace by steps of the protocol. Therefore, we define six actions modeling the sending and the receiving of a particular message. In Figure 9.1 we illustrate the Gets event of a corresponding Says event with an additional star *. In the eSSCM we consider a pair of events, e.g. sent1 and rcvd1, according to each protocol step. We already defined the pairs in StateLabel. From the point of view of the smart card we have to model six actions divided into sending and receiving actions. A protocol run starts with the receiving of the message askRandom. Hence, component CommC is modeled to stay in an idle mode waiting for requests represented by the local state ready. This is defined in the INITIAL slot of the CommC behavior specification given in Appendix C.2. In the following we exemplary explain the first two actions of component CommC.

The first action receive1 can be taken if the schedule variable next is set to participant and if the message (askRandom,AP) has been received via channel sc_in from an external application AP. Then there shall exist a cer-

tain pair of smart card SC and external application AP out of the set of all smart cards and external applications so that the local state list of component CommC represented by SC holds the initial entry ready.

```
EX SC, AP, X:
inState(stateInfo(ready), select(StateSC, SC))
```

Please note, the array StateSC holds the local traces of all components CommC represented by SC. The function select(StateSC, SC) determines the particular local trace $\texttt{Stateentrylist}_{\texttt{SC}}$ of agent SC. The predicate inState gives the existence of a particular entry StateLabel in the local trace.

Moreover, the global state of all protocol runs represented by net shall also hold the already received message, which says that someone (represented by agent X) sent the message in the past. If the conditions hold we update the local state by saying that message 1 has been successfully received from an external application AP. The state label rcvd1(AP) is added to the local state list. We additionally update the global state net by removing the entry (X,SC,askRandom). Lastly, the variable next is set to observer, which is discussed in the next section. If the conditions do not hold all the variables mentioned in UNCHANGED remain unchanged.

```
receive1 ::=
    IF next = participant AND
       sc_in = pair(num(askRandom), agent(friend(AP)))
    THEN EX SC, AP, X :
       (SC <= max AND
        inState(stateInfo(ready), select(StateSC, SC)) AND
        netmember(mknetentry(X, SC, pair(num(askRandom),
                             agent(friend(AP)))), net) AND
        AP <= max AND
        StateSC' = enterstateentry(SC, stateInfo(rcvd1(AP)),
                   delstateentry(SC, stateInfo(ready), StateSC)) AND
        net' = deletenetentry(mknetentry(X, SC, pair(num(askRandom),
                              agent(friend(AP)))), net)) AND
        next' = observer AND
        UNCHANGED(key_list,h2_scout,sc_out,Rsp,Chlng,n_sc,k_apsc)
    ELSE UNCHANGED(key_list,next,net,sc_out,h2_scout,StateSC,
                   Rsp,Chlng,n_sc,k_apsc)
    FI
```

The action send2 of component CommC represents step 2 of the mutual authentication protocol. If the schedule variable next is set to participant and there are key data left in the buffer key_list then a new nonce can be send. Please note, we do not specify a special nonce buffer here and interpret keys

as nonces. To send a new nonce it is first checked if the local state is set to the received message in step 1. Then, a new nonce is taken from the buffer, deleted in the buffer, sent via output channel `sc_out` and assigned to the internal variable `Chlng`. This is necessary, because we have to remember the challenge in subsequent steps. The performed action is then added to the local state by adding `sent2(SC,AP)`. Moreover, the auxiliary variables `h2_scout` and `net` are updated and the schedule variable `next` is set to `observer`.

```
send2 ::=
  IF next = participant AND
     key_list /= nil
  THEN EX SC, AP :
     (SC <= max AND
      inState(stateInfo(rcvd1(AP)), select(StateSC, SC)) AND
      n_sc' = last(key_list) AND
      Chlng' = nonce(n_sc') AND
      key_list' = butlast(key_list) AND
      sc_out' = pair(agent(friend(AP)), nonce(n_sc')) AND
      h2_scout' = cons(n_sc', h2_scout) AND
      StateSC' = enterstateentry(SC, stateInfo(sent2(AP, Nsc)),
                 delstateentry(SC,stateInfo(rcvd1(SC)),StateSC)) AND
      net' = Bnetlist.addnetentry(mknetentry(SC, AP, Nsc), net)) AND
      next' = observer AND
      UNCHANGED(Rsp, k_apsc)
  ELSE UNCHANGED(key_list, next, net, sc_out, h2_scout, StateSC,
                 Rsp, Chlng, n_sc, k_apsc)
  FI
```

In this way we model all steps of the mutual authentication protocol in terms of actions for the components `CommC` and `ExtAppl`. The complete specification is given in Appendix C.2.

Lastly, we have to define the behavior of `CommC`, whereby we want allow behaviors that involve parallel runs of protocol sessions with different external applications. Because we put all necessary preconditions in the actions we do not need to define additional control flow in the `SPEC` slot of the component. There might be several actions enabled at a time but there is only one action enabled for a particular protocol run. The distinction between receiving and sending actions allows the handling of parallel protocol sessions. Hence, component `CommC` is not forced to immediately take the enabled action `send2` but may take the also enabled action `receive1` of a new parallel protocol session. We already gave an illustrative example of protocol runs in Figure 9.1. all actions of the components are stated in the `TRANSITION` slot non-deterministically.

```
SPEC
   TRANSITIONS [receive1, send2, receive3, send4, receive5, send6]
               {key_list, nonces_list, StateSC, Chlng, Rsp, sc_out,
                h2_scout, n_sc, k_apsc}
```

In this fashion we specified all protocol steps for components `CommC` and `ExtAppl` acting on behalf of agents SC and AP of the mutual authentication protocol, respectively. We do not have to model the attacker again in the eSSCM because the attacker is considered in the inductive model. All we have to show here is that the verified protocol is implemented correctly by the smart card represented by component `CommC`. The correctness proof is done with an additional component `observer` implementing the observer mapping functionality.

9.3.2 Observer component

Since we want to verify that every sequence of actions taken by components `CommC` and `ExtAppl` corresponds to a valid *map* trace, we specify an additional component `TLSPEC Observer` having as input the global state of all protocol runs given in the variable `SHARED INOUT net: Netentrylist`. The output `trace` of the observer is a trace of type `ProtocolTrace` defined in the inductive protocol model. As we will see later on, the predicate `MAP(ProtocolTrace)` also defined in the inductive model shall always validate true for this very trace. The variable `OUT Obsnet: Netentrylist` is used by the component to detect whether the environment performed a protocol relevant step or an other step. This ensures that the observer does not recognize a protocol step twice.

```
TLSPEC Observer
 USING Definition
 DATA
    SHARED INOUT next : Component   /*Scheduling*/
    OUT trace : ProtocolTrace       /*Paulson protocol trace*/
    SHARED INOUT net : Netentrylist /*detect events "Says" "Gets"*/
    OUT Obsnet : Netentrylist       /*detecting changes of "net"*/
```

The Development Graph of the eSSCM given in Figure 9.10 shows the additional observer component. Because all components are modeled concurrently we must ensure that the observer component records all communication events between `CommC` and `ExtAppl`. This makes the introduction of a control flow necessary, which guarantees that after one of the components `CommC` or `ExtAppl` performed a step the observer is forced to perform the subsequent step. This is implemented by the variable `SHARED INOUT next`

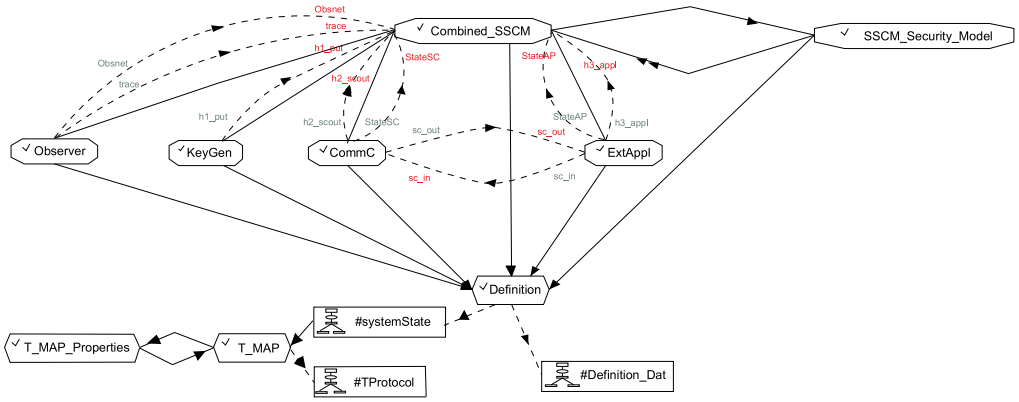


Figure 9.10: VSE Development Graph of the extended SSCM

that is set to `observer` by components `CommC` and `ExtAppl` after performing a step and set to `participants` after the observer took a step. If `next` is set to `participant` component `CommC` or `ExtAppl` may perform a step non-deterministically.

This restriction of behaviors defined by the schedule variable is part of the auxiliary specification only and will not be refined in a further refinement step. We just use it for verification purposes. It does not effect the behavior of the entire system, because it just adds steps of the observer immediately after the steps taken by `CommC` and `ExtAppl` in a certain system behavior.

The observer component is specified to have one single action `buildtrace` that is enabled if the mentioned schedule variable `next` is set to `observer`. The action first checks if the global state of the protocol run changed meaning the step previously taken by one of the other components affected the protocol run. This is modeled by checking `Obsnet = net`. If it holds the observer leaves the variable `trace` unchanged. Otherwise it has to check if a `Says` or a `Gets` event occurred. The agents add an entry to the variable `net` if they sent a message; they delete an entry if they successfully received a message. A `Says` event of the inductive model corresponds to the addition of entries and a `Gets` event corresponds to the deletion of entries.

$$\begin{array}{llll} \text{if} & \texttt{net} = \texttt{Obsnet} \cup \{(\texttt{X}, \texttt{Y}, \texttt{M})\} & \text{then} & \texttt{Says}(\texttt{X}, \texttt{Y}, \texttt{M}) \\ \text{if} & \texttt{net} = \texttt{Obsnet} \setminus \{(\texttt{X}, \texttt{Y}, \texttt{M})\} & \text{then} & \texttt{Gets}(\texttt{Y}, \texttt{M}) \end{array}$$

This is implemented by checking if a certain entry of the global state list `net` is an `addnetentry()` or a `deletenetentry()`. In the former case the action adds the corresponding `Says` event to the actual protocol trace `trace` and in the latter case it adds the corresponding `Gets` event. Please remember, a protocol trace in the inductive model consists of `Says`, `Notes` and `Gets` events only (see Section 8.1.1). The specification of the action is given below.

```
buildtrace ::=
    next = observer AND
    next' = participant AND
    ((Obsnet = net ->
      evtP = F) AND
     (Obsnet /= net ->
      (evtP = T AND
       (EX X, Y, M :
        (net = Bnetlist.addnetentry(mknetentry(X,Y,M),Obsnet) AND
         newEv = says(friend(X), friend(Y), M)) OR
        (net = deletenetentry(mknetentry(X, Y, M), Obsnet) AND
         newEv = gets(friend(Y), M)))))) AND
    (evtP = T ->
     trace' = addEvent(newEv, trace)) AND
    (evtP = F ->
     trace' = trace) AND
    Obsnet' = net
```

The behavior of the component `Observer` is trivial saying the action is performed or the variables `trace` and `Obsnet` remain unchanged. We already mentioned the new proof obligation regarding the variable `trace` that we discuss in more detail in the next section.

```
SPEC INITIAL trace = nullEvent
      TRANSITIONS [buildtrace] {trace, Obsnet}
```

Once again, the observer component and the variables `trace`, `Obsnet` and `net` are all together part of the auxiliary specification modeling history knowledge of the behavior of the system. Because we use this knowledge to verify safety properties at this abstract level, it will not be refined in further development steps.

9.4 New verification tasks in VSE-II

The additional component `Observer` records all messages sent via channels `sc_in` and `sc_out`. Since every sequence of sent messages corresponds to a certain behavior of the entire system we want to make sure that each sequence additionally corresponds to a valid *map* trace. Hence, we have to show that the observed trace represented by the variable `trace` always yields true for the predicate `MAP` for all behaviors of the entire system. This leads to the new safety property: $\Box$(`MAP(S_S_C_M.trace)`) specified in the security model of the extended Simple Smart Card Model `TLSPEC SSCM_Security_Model`. The new *Protocol Property* is added to the already verified safety properties discussed in Section 7.3.4.

```
TLSPEC SSCM_Security_Model
 USING Definition
 INCLUDE S_S_C_M = Combined_SSCM
 SPEC
   /* confidentiality property: */
   [] (S_S_C_M.h2_scout /= nil ->
       NOT p_check(first(S_S_C_M.h2_scout)));
   /* integrity property: */
   [] (S_S_C_M.h3_appl /= nil ->
       p_check(first(S_S_C_M.h3_appl)));
   /* non-duplicating property: */
   [] (S_S_C_M.h3_appl /= nil ->
       greatest_elem(first(S_S_C_M.h3_appl),rest(S_S_C_M.h3_appl)));
   /* protocol property: */
   [] MAP(S_S_C_M.trace)
TLSPECEND
```

To prove the Protocol Property for the component `SSCM_Security_Model` we first have to prove the property for the combined system represented by component `Combined_SSCM`.

$$\Box(\texttt{MAP}(\texttt{Combined\_SSCM.trace})) \tag{9.1}$$

Please remember, the predicate `MAP` is inductively defined in the model of the mutual authentication protocol in the development object `THEORY T_MAP` as discussed in Section 8.3.1. Hence, we have to show that the predicate `MAP` holds for the empty trace and for all possible changes of the variable. The proof of the initial case is trivial, because it is given by the definition of the predicate in the inductive model.

$$\Box(\texttt{trace} = \texttt{nullEvent} \rightarrow \texttt{MAP}(\texttt{trace})) \tag{9.2}$$

Due to readability we assume variable `trace` to be the abbreviation of variable `Combined_SSCM.trace` in the following. As a helping lemma we consider a further trivial case stating that a valid trace remains valid if it does not change its value while an arbitrary action is taken.

$$\Box((\texttt{MAP}(\texttt{trace}) \wedge \texttt{trace}' = \texttt{trace}) \rightarrow \texttt{MAP}(\texttt{trace}')) \tag{9.3}$$

The change of a valid trace is specified by its extension with a new event. The resulting extended trace should also be a valid *map* trace.

$$\Box((\texttt{MAP}(\texttt{trace}) \wedge \texttt{trace}' = \texttt{addEvent}(\texttt{newEv}, \texttt{trace})) \rightarrow \texttt{MAP}(\texttt{trace}')) \tag{9.4}$$

The behavior of the entire system represented by `Combined_SSCM` is determined by the interleaved behaviors of all components. Because the variable `trace` is affected by the observer only, we can prove Lemma 9.1 by considering the full behavior of the observer and by applying Lemmas 9.2, 9.3 and 9.4. This behavior indeed leaves `trace` unchanged if the environment takes a step (it is stated in the stuttering index) or if the global state of all protocol runs did not change, which leaves the shared variable `net` unchanged. The other case in the behavior of the observer is the adding of a new observed event to the actual protocol trace, which is either a `Says` or a `Gets` event. There are no other actions changing the variable `trace` in a different way.

The proof of Lemma 9.4 requires the insertion of the inductive definition of the predicate `addEvent` given in the inductive model.

$$
\begin{aligned}
&\texttt{MAP}(\texttt{addEvent}(\texttt{newEv}, \texttt{trace})) \leftrightarrow \\
&\texttt{MAP}(\texttt{trace}) \wedge \\
&(\texttt{MAP\_Says1}(\texttt{newEv}, \texttt{trace}) \vee \texttt{MAP\_Says2}(\texttt{newEv}, \texttt{trace}) \vee \\
&\ \texttt{MAP\_Says3}(\texttt{newEv}, \texttt{trace}) \vee \texttt{MAP\_Says4}(\texttt{newEv}, \texttt{trace}) \vee \\
&\ \texttt{MAP\_Says5}(\texttt{newEv}, \texttt{trace}) \vee \texttt{MAP\_Says6}(\texttt{newEv}, \texttt{trace}) \vee \\
&\ \texttt{MAP\_Oops}(\texttt{newEv}, \texttt{trace}) \vee \\
&\ \texttt{Fake\_event}(\texttt{newEv}, \texttt{trace}) \vee \\
&\ \texttt{Gets\_event}(\texttt{newEv}, \texttt{trace}))
\end{aligned}
\tag{9.5}
$$

Applying Lemma 9.5 leads to a case distinction for every predicate according to the disjunction. Each predicate represents either a certain step in the protocol or steps taken by the attacker. All predicates used in Lemma 9.5 are also defined in the inductive model. For instance, the predicate `MAP_Says1` denotes the first step of the mutual authentication protocol, which is the sending of an `askRandom` message from the external application (here represented by `ExtAppl`) to the smart card (represented by `CommC`). Predicate `MAP_Says2` denotes the sending of the nonce `Nsc` from the smart card to the external application and so forth. Because we are mainly concerned with the security policy of the smart card, we concentrate on the component `CommC`.

The proof of each single case follows the same structure and we exemplary give the proof sketch of predicate `MAP_Says2` that is defined in the inductive model as follows.

$$
\begin{aligned}
&\texttt{MAP\_Says2}(\texttt{newEv}, \texttt{trace}) \leftrightarrow \\
&\texttt{Nsc} \notin \texttt{used}(\texttt{trace}) \wedge \\
&\texttt{Gets}(\texttt{SC}, (\texttt{askRandom}, \texttt{AP})) \in \texttt{trace} \wedge \\
&\texttt{newEv} = \texttt{Says}(\texttt{SP}, \texttt{AP}, \texttt{Nsc})
\end{aligned}
\tag{9.6}
$$

Lemma 9.6 says that whenever the event `Says(SC, AP, Nsc)` occurred the event `Gets(SC, (askRandom, AP))` must have occurred before and the used nonce should be fresh. Therefore, we prove the following helping lemma stating that it should hold for every possible behavior of the system: if the mentioned `Says` event is added to variable `trace` then variable `net` must have contained the element `(SC, AP, Nsc)`.

$$\begin{aligned}\Box(\ & \mathtt{trace}' = \mathtt{addEvent}(\mathtt{Says}(\mathtt{SP}, \mathtt{AP}, \mathtt{Nsc}), \mathtt{trace}) \rightarrow \\ & \mathtt{net} = \mathtt{Obsnet} \cup \{(\mathtt{SC}, \mathtt{AP}, \mathtt{Nsc})\} \quad)\end{aligned} \tag{9.7}$$

The proof of the lemma is locally done to the behavior of the observer, because no other component accesses the variable `trace`. As explained in the previous section the action `buildtrace` of the observer is designed to add says events only if the corresponding element is in the variable `net`. Next, we have to proof a lemma saying that the element `(SC,AP,Nsc)` of list `net` could have only been added if component `CommC`, which is instantiated by `SC`, received the message `(askRandom, AP)` of step 1 of the protocol before. In terms of the specification of `CommC` the `StateLabel` representing the receiving of the first message must be in the local history list of the certain representative `SC` denoted as $\mathtt{StateSC}_{\mathtt{SC}}$ of type `Stateentrylist`.

$$\begin{aligned}\Box(\ & \mathtt{net}' = \mathtt{Obsnet} \cup \{(\mathtt{SC}, \mathtt{AP}, \mathtt{Nsc})\} \rightarrow \\ & \mathtt{rcvd1}(\mathtt{AP}) \in \mathtt{StateSC}_{\mathtt{SC}} \quad)\end{aligned} \tag{9.8}$$

In fact, there is only one action `send2` of the entire specification of the eSSCM that adds the element `(SC, AP, Nsc)` to the list `net`. The action `send2` of the behavior of component `CommC` is specified to fire only if action `rcvd1` has previously been taken. This is modeled by the entry `rcvd1(AP)` in the local state $\mathtt{StateSC}_{\mathtt{SC}}$ of a certain smart card `SC`.

Moreover, if action `receive1` has already been taken then either the following action `buildtrace` by the observer has been taken or the observer is ready to take the action. Please note, we introduced the control flow variable `next` to ensure that the observer takes a step immediately after one of the components `CommC` and `ExtAppl` took a step. Hence, we can prove the following helping lemma.

$$\begin{aligned}\Box(\ & \mathtt{rcvd1}(\mathtt{AP}) \in \mathtt{StateSC}_{\mathtt{SC}} \rightarrow \\ & (\mathtt{Gets}(\mathtt{SC}, (\mathtt{askRandom}, \mathtt{AP}) \in \mathtt{trace}) \vee \\ & (\mathtt{next} = \mathtt{observer} \wedge \mathtt{net} \neq \mathtt{Obsnet} \wedge \\ & \mathtt{net} = \mathtt{Obsnet} \setminus \{(\mathtt{AP}, \mathtt{SC}, (\mathtt{askRandom}, \mathtt{AP}))\})) \quad)\end{aligned} \tag{9.9}$$

We come back to the proof of Lemma 9.6 and the corresponding case of Lemma 9.5. The application of Lemmas 9.7, 9.8 and 9.9 proves that whenever

the event `Says(SC, AP, Nsc)` has been sent and thus is element of variable `trace` the event `Gets(SC,(askRandom,AP))` must have previously been sent and is also element of `trace`. It remains to show that the new nonce taken from the buffer `key_list` has not been used in previous protocol sessions. Because we already proved the non-duplicating property for key data k it holds: $\forall k : k \in \texttt{key\_list} \rightarrow k \notin \texttt{used}(\texttt{trace})$. Applying this lemma closes the proof of the validity of predicate `MAP_Says2`.

Hence, we proved that for all behaviors of the entire system the extension of a valid trace `map(trace)` with a new event `Says(SC, AP, Nsc)` results a valid trace. It is the VSE-II system that ensures correctness: if we perform a proof locally to the behavior of a component, e.g. Lemma 9.7, from the perspective of the combined system then there is no other component affecting the particular proof. Otherwise we could not have proven the particular lemma.

In this way all cases of the proof of Lemma 9.4 have to be performed. We just described the case `MAP_Says2`. From the perspective of the security policy of the smart card the cases `MAP_Says4` and `MAP_Says6` and `Gets_event` are of interest. The cases `MAP_Says1`, `MAP_Says3` and `MAP_Says5` belong to actions taken by the external application. Performing the proofs and providing the verified specification of an external application is an excellent starting point for the further development of an external application. Leaving the cases open in the proof can be interpreted as an assumption that component `ExtAppl` behaves as expected by the system.

Moreover, with the given specification of the smart card we are not able to close the cases `MAP_Oops` and `Fake_event`. This is due to the fact that there are no actions generating arbitrary messages. More precisely, with an additional verified component `ExtAppl` we can even exclude the existence of "Oops" cases. It is the aim of verifying safety properties to show that there is no other behavior than the expected one. In this way we discharge the assumption made in the proof of the Confidentiality Lemma 8.3 of the session key in the protocol model by saying that the "Oops" case can not occur.

9.5 Conclusion

The aim of this chapter is to link safety and security aspects of a system. In Chapter 7 we discuss safety properties of a state based Simple Smart Card Model using the temporal logic of actions. For instance, one of the safety properties states that the transformation function T is always applied on sent key data. Because we argue the expressiveness of the verified model, we consider a refinement of the transformation function T in form of a cryptographic protocol. This leads to two new verification tasks of showing that

a particular protocol indeed fulfills the requirements of the transformation function and that the found instantiation is correctly implemented by the system model. The former task is done in Chapter 8 by verifying security properties of a mutual authentication protocol with the inductive approach by Paulson. The latter task is done in this chapter by applying an observer methodology. In this way we show that the Communication Component of the Extended Simple Smart Card Model (eSSCM), which has been enhanced by the protocol steps, behaves exactly as the corresponding agent in the verified protocol model for each behavior of the entire system model.

The proof is performed by means of an additional component `Observer` recording all messages sent via output channel `sc_out` and input channel `sc_in` of the Extended Simple Smart Card Model. With the recorded information the `Observer` builds a trace as defined in the inductive protocol model. Hence, it has to be shown that the built trace is always a valid protocol trace. Since we perform all specifications and proofs in VSE-II the consistency between the two models is guaranteed. It must be underlined that we do not have to prove security properties in the eSSCM again, because the correctness of the protocol in the presence of an attacker has already been verified in the inductive model. In this way it has been shown that the eSSCM satisfies both the safety properties and the inductive model of the mutual authentication protocol and thus the security properties. This is a new level of quality in specifying and verifying security policies.

But the specification of a security policy denotes the first step in the overall development of the system if we follow the waterfall model (Roy87). The stepwise refinement of the abstract system specification is logically and in the *safety* sense an implication. We say that all behaviors of the more abstract level are also behaviors of the refined level and thus, safety properties are preserved under refinement.

In opposition, the formal analysis of cryptographic protocols assumes the used *cryptographic primitives* to be *perfect*, e.g. symmetric and asymmetric algorithms or hash functions. For instance, in this sense perfect means that decrypting a cipher text is possible only with the corresponding key. This is a very strong assumption, because in the analysis of cryptographic primitives polynomial bounded attacker models are considered. The Backes-Pfitzmann-Waidner Model tries to bridge this gap (BPW03). Hence, the refinement of a cryptographic protocol with particular cryptographic primitives may require the change of the attacker model (San06), which is also known as the *refinement paradox* (Jür01). Nevertheless, the formal analysis of cryptographic protocols with traditional attacker models proved very helpful so far and is a very good start for the development of dependable systems.

Chapter 10

Summary and conclusion

The dependable development of security critical systems is an important research field, as building a security system is more complex than just adding some cryptography (like the encryption or signing of data). Today, it is commonly accepted that the specification of a security policy, which has to be implemented by a certain security system, is a very good start point for those developments and has been strongly influenced by the publication of international security evaluation criteria like the Common Criteria. The quality of the underlying security policy, which deals with the main security functions implementing the desired security objectives, ultimately determines the quality of the security product.

This work focuses on the specification and verification of security policies in the context of multi-applicative smart cards as they are becoming indispensable in our daily life. Opposite to traditional smart cards, multi-applicative smart cards are able to hold several applications on the card that may be totally isolated or may communicate to each other within the card. A few multi-applicative smart cards readily implement different security policies, for instance Java Card, Multos, SMaCOS or BasicCard. All proposed smart card operating systems provide different mechanisms for confidentiality and integrity requirements between on-card applications. Security requirements regarding the communication between an on-card application and external applications or external devices are not addressed by these policies but are demanded by modern applications, as shown in this work with four representative case studies.

We proposed a security policy of a smart card operating system that additionally controls the communication between on-card and external applications. The operating system acts as a firewall between the on-card applications and also between the on-card and external applications or devices. We extended the security policy proposed by SMaCOS with further secu-

rity objectives and security functions regarding the integration of external applications and devices.

The integration required additional mechanisms for the authentication of external applications or devices and mechanisms for the secure transmission of data. Access rights of all applications and devices are determined by secrecy and integrity levels of the Bell/LaPadula and Biba model. As both models have been regarded as less relevant for real world scenarios, we exemplary used the case study `BioSig` (electronic signatures and biometrics) to demonstrate the applicability of the extended security policy. In our hands, integrity levels and categories proved very suitable to map evaluation levels of the Common Criteria and ITSEC in a very flexible way. The proper definition of access classes and categories as well as the proper assignment of all applications and devices involved must be done in an off-card process. This may lead to new security tasks, for instance the cascade problem of multi-level security systems.

Altogether, with the extended security policy we achieved confidentiality and integrity between on-card applications with access control mechanisms as all communications are under the full control of the operating system. The external communication takes place over an open network and can no longer be controlled by the operating system. Confidentiality and integrity have been achieved by means of cryptographic protocols. Generally, the proposed methodology of integrating external applications and devices is not limited to the SMaCOS security policy but can basically be applied to all existing policies. A further promising approach are role-based security policies. In contrast to the fixed mandatory access control mechanisms above mentioned they provide predefined roles with specific access rights assigned to an entity, for instance a user or an application. In this way, requirements on the separation of duty for a certain entity can be implemented.

Since the quality of a security system countering the identified security threats is determined by the underlying security policy, its analysis becomes a key issue in the development process. We therefore considered security threats, objectives and functions in a triangle relation that we called the Security Triangle, whereby the functions and objectives are part of the security policy. Although those relations are often individually recognized the term Security Triangle is rarely used but allows a more comprehensive view on the topic. The analysis of the Security Triangle is central for the quality of a security product and leads to three questions also defining quality parameters of security policies: `Q1` Effectiveness, `Q2` Completeness and `Q3` Consistency of the security policy.

The more precise these parameters can be defined the higher the quality of a security product. Formal methods can give unambiguous answers to the

questions, whereby two main fields of applying formal methods can be identified: formal models of access control or information flow and formal analysis of cryptographic protocols. While the former methods address questions `Q2` Completeness and `Q3` Consistency, the latter are concerned with attacker models and address question `Q1` Effectiveness. To distinguish properties to be verified in the different fields we refer to the former as safety properties and the latter as security properties.

At present, verifying a security policy is commonly associated with the verification of the used access control model or the used information flow model, which generally leads to the verification of safety properties. Because we aimed at the formal verification of the entire extended security policy, we have been confronted with both safety and security properties.

In this work we provided a framework that links state-based specifications of an abstract system in the Temporal Logic of Actions by Lamport with inductive models of cryptographic protocols by Paulson using an observer methodology. We performed all specifications and proofs in the development tool Verification Support Environment (VSE-II). Therefore, we considered a Simple Smart Card Model (SSCM) that concentrates on the external communication only and verified safety properties, for instance stating that certain data is transferred confidentially. We verified that an abstract transformation function is always applied on the transferred data for all possible behaviors of the entire system. As already mentioned, the verification of security policies usually stops at this stage assuring that it is complete and consistent.

We further strengthened the expressiveness of the extended security policy by refining the abstract transformation function in the SSCM with a concrete cryptographic protocol. This led to two new verification tasks, which show that a particular protocol indeed fulfills the requirements of the transformation function and that the found instantiation is correctly implemented by the system model. For the first task, we verified security properties of a mutual authentication protocol with the inductive approach by Paulson. For the second task, we applied an observer methodology.

We showed that the Extended Simple Smart Card Model (eSSCM), which has been enhanced by the protocol steps, behaves exactly as the corresponding agent in the verified protocol model for each behavior of the entire system model. Since we performed all specifications and proofs in VSE-II the consistency between the two models is guaranteed. It must be underlined that we did not have to prove security properties in the eSSCM again, because the correctness of the protocol in the presence of an attacker has already been verified in the inductive model.

In summary, this work shows that the eSSCM satisfies both the safety properties and the security properties. The SSCM and the eSSCM represent

security policy models sharing the same security objectives. They both differ in the security functions implementing the security objectives, whereby the eSSCM is much more expressive than the SSCM. The eSSCM specifies more detailed functionality implementing the security objectives and additionally verifies the effectiveness of the security policy, because the eSSCM takes a formal attacker model into account. Hence, it allows conclusions on a formal basis about the effectiveness of the chosen security functions and thus, about the effectiveness of the security policy counterfighting identified security threats. This is a new and more advanced level of quality in specifying and verifying security policies.

The specification of a security policy and the verification of its formal model denote the first steps in the overall development process of a system. Further steps involve stepwise refinements of the abstract system specification given that verified safety properties are preserved under refinement. Although the formal analysis of cryptographic protocols assumes perfection of the cryptographic primitives (like symmetric and asymmetric algorithms or hash functions), this strong assumption does not hold in practice, which is commonly known as the refinement paradox.

Taken together, the formal analysis of security policies including cryptographic protocol analysis proved very suitable for the proposed model and is a good start for the dependable development of security systems. The formal modeling and analysis work done in the tool VSE-II turned out to be valuable, since it facilitated the maintenance of consistency and proof obligations while editing the model. In addition, this tool also assists in the development of further refinement steps, because the assumptions made on functions and data in the abstract model become requirements in the refined specification levels. Since the tool VSE-II is additionally approved for higher level evaluations under the Common Criteria by the German Federal Office for Information Security (BSI)[1], its usage contributes to an improved cost-benefit ratio, especially if further development cycles in the context of new product releases and reevaluations are foreseen. This has been practically experienced, as an instantiation of the generic SSCM was used in an industrial evaluation project.

[1] http://www.bsi.de

Appendix A

State-based specification in VSE

A.1 Model of the SSCM in VSE-SL

```
1. Combined_SSCM
================

TLSPEC Combined_SSCM
 PURPOSE
 "Defines all connections of the components
  and the entire system "
 USING Definition
 DATA
      /* history variable of the channel put */
      OUT h1_put : list
      /* history variable of the channel sc_out */
      OUT h2_scout : list
      /* history variable of all external applications */
      OUT h3_appl : list
 COMBINE KeyGen [KeyGen.h1_put -> Combined_SSCM.h1_put] ;
         CommC [CommC.h2_scout -> Combined_SSCM.h2_scout,
                CommC.auth <- ExtAppl.auth]
         SHARED [CommC.key_list <- KeyGen.key_list];
         ExtAppl [ExtAppl.h3_appl -> Combined_SSCM.h3_appl,
                  ExtAppl.tauth <- CommC.tauth,
                  ExtAppl.tkey <- CommC.tkey]
 SATISFIES SSCM_Security_Model
TLSPECEND

2. CommC
========
```

```
TLSPEC CommC
 PURPOSE
 " Communication Component of the Smart Card "
 USING Definition
 DATA
      /* channel get from buffer */
      SHARED INOUT key_list : list
      /* channel sc_in from ExtAppl */
      IN auth : nat
      /* channel sc_out to ExtAppl */
      OUT tkey : nat
      /* actual authentication data */
      OUT tauth : nat
      /* history variable of channel sc_out */
      OUT h2_scout : list
 ACTIONS
 /* after successful verification of the authentication data,
      action takes the last (oldest) key from the key_list,
      transforms it and deletes it */
 checked_transform (value : IN nat)::=
     IF v_check(value) AND
        key_list /= nil
     THEN tauth' = value AND
          tkey' = trans(last(key_list)) AND
          key_list' = butlast(key_list) AND
          h2_scout' = cons(tkey', h2_scout)
     ELSE UNCHANGED(tauth, tkey, key_list, h2_scout)
     FI
 SPEC
      /* Specification of the behavior of the seller */
      INITIAL BEGIN
               tauth := 0;
               tkey := 0;
               h2_scout := nil
              END
      TRANSITIONS BEGIN
                   WHILE TRUE DO
                    IF auth /= tauth
                    THEN checked_transform(auth)
                    FI
                   OD
                  END {key_list, tkey, tauth, h2_scout}
TLSPECEND

3. Definition
=============

THEORY Definition
 USING Channel_Data_LQs
```

```
 FUNCTIONS inv_trans : nat -> nat;
           trans : nat -> nat
 PREDICATES v_check : nat;
            p_check : nat;
            p_check_list : list;
            greatest_elem : nat, list
 VARS k, e : nat;
      l : list
 AXIOMS FOR trans : p_check(k) ->
                    NOT p_check(trans(k))
        FOR p_check_list : p_check_list(l) <->
                           l = nil OR
                           (p_check(first(l)) AND
                            p_check_list(rest(l)))
        FOR p_check_list : p_check_list(l) ->
                           p_check_list(butlast(l))
        FOR p_check_list : (p_check_list(l) ->
                            p_check(last(l)))
        FOR inv_trans : k = inv_trans(trans(k))
        FOR greatest_elem : (greatest_elem(k,l) <->
                             l = nil OR
                             (k > first(l) AND
                              greatest_elem(k,rest(l))))
THEORYEND

4. ExtAppl
==========

TLSPEC ExtAppl
 PURPOSE
 " External Application receiving a key from CommC "
 USING Definition
 DATA INTERNAL xc : nat
      /* send key request and authentication data
         to CommC via channel sc_in */
      OUT auth : nat
      /* history variable of received keys */
      OUT h3_appl : list
      /* receive transformed keys from channel sc_out */
      IN tkey : nat
      /* CommC current authentication data */
      IN tauth : nat
 ACTIONS
 send_new_req ::= auth' = auth + 1;
                  UNCHANGED(xc, h3_appl)
 get_key ::= IF tkey /= 0
             THEN xc' = inv_trans(tkey) AND
                  h3_appl' = cons(xc', h3_appl) AND
                  UNCHANGED(auth)
```

```
            ELSE UNCHANGED(xc, auth, h3_appl)
            FI
 SPEC INITIAL BEGIN
               auth := 0;
               xc := 0;
               h3_appl := nil
              END
      TRANSITIONS BEGIN
                   WHILE TRUE DO
                    send_new_req;
                    IF tauth = auth
                    THEN get_key
                    FI
                   OD
                  END {xc, auth, h3_appl}
 HIDE xc
TLSPECEND

5. KeyGen
=========

TLSPEC KeyGen
 PURPOSE
 " Generation of keys "
 USING Definition
 DATA
      /* channel put to buffer */
      SHARED INOUT key_list : list
      /* history variable of channel put */
      OUT h1_put : list
      /* internal computed random value */
      INTERNAL xp : nat
 ACTIONS
 /* Specification of the compute action.
     The computation is left unspecified. */
 generate ::= xp' = xp + 1;
              UNCHANGED(key_list, h1_put)
 /* specification of the predicate and the send function */
 check_send ::= IF p_check(xp)
                THEN key_list' = cons(xp, key_list) AND
                     h1_put' = cons(xp, h1_put) AND
                     UNCHANGED(xp)
                ELSE UNCHANGED(xp, key_list, h1_put)
                FI
 SPEC
      /* Specification of the behavior of component KeyGen */
      INITIAL BEGIN
               xp := 1;
               h1_put := nil;
```

```
              key_list := nil
             END
     TRANSITIONS BEGIN
                  WHILE TRUE DO
                   generate;
                   check_send
                  OD
                 END {key_list, h1_put, xp}
 /* Variables not externally visible */
 HIDE xp
TLSPECEND
```

A.2 Properties of the SSCM in VSE-SL

```
6. SSCM_Security_Model
======================

TLSPEC SSCM_Security_Model
 PURPOSE
 " Specification of the security model of the entire system "
 USING Definition
 INCLUDE S_S_C_M = Combined_SSCM
 SPEC
   /* confidentiality property:
      all transmitted keys from CommC to ExtAppl are
      readable for the authenticated ExtAppl only, or
      a received key can never occur in the delivery channel
          */
   [] (S_S_C_M.h2_scout /= nil ->
       NOT p_check(first(S_S_C_M.h2_scout)));
   /* integrity property:
      any delivered key has not been changed*/
   [] (S_S_C_M.h3_appl /= nil ->
       p_check(first(S_S_C_M.h3_appl)));
   /* non-duplicating property:
      any delivered key can not appear twice in consumer's
      output list (or in terms of the model: the first element
      in the list is always the greatest element of the list) */
   [] (S_S_C_M.h3_appl /= nil ->
       greatest_elem(first(S_S_C_M.h3_appl),rest(S_S_C_M.h3_appl)))
TLSPECEND
```

Appendix B

VSE model of the authentication protocol

B.1 Model of the protocol in VSE-SL

```
THEORY T_MAP
 USING TProtocol
 FUNCTIONS Adm : AgentT;
           askRandom, getSessKey : nat
 PREDICATES MAP : ProtocolTrace;
            MAP_Says1, MAP_Says2, MAP_Says3, MAP_Says4, MAP_Says5,
            MAP_Says6, MAP_Oops : ProtocolEvent, ProtocolTrace
 VARS ev : ProtocolEvent;
      evs : ProtocolTrace;
      SC, AP : AgentT;
      Nsc, Nsc2, Nap : nat;
      Kapsc : KeyT
 AXIOMS
   secureAg0 : Adm = secureAg(0);
   notEqaskRandom_getSessKey : NOT askRandom = getSessKey;
   MAPNull : MAP(nullEvent);
   MAPAdd : MAP(addEvent(ev,evs)) <->
            (MAP(evs) AND
             (MAP_Says1(ev,evs) OR
              MAP_Says2(ev,evs) OR
              MAP_Says3(ev,evs) OR
              MAP_Says4(ev,evs) OR
              MAP_Says5(ev,evs) OR
              MAP_Says6(ev,evs) OR
              MAP_Oops(ev,evs) OR
              Fake_event(ev,evs) OR
              Gets_event(ev,evs)));
   /* Step1 : */
```

```
MAPSays1 : MAP_Says1(ev,evs) <->
           EX AP, SC :
           (NOT isSecure(AP) AND
            NOT isSecure(SC) AND
            NOT (SC = AP) AND
            ev = Says(AP,SC,pair(num(askRandom),agent(AP))));
/* Step2 : */
MAPSays2 : MAP_Says2(ev,evs) <->
           EX SC, AP, Nsc :
           (NOT msgIN(nonce(Nsc),used(evs)) AND
            eventIN(Gets(SC,pair(num(askRandom),
                         agent(AP))),evs) AND
            ev = Says(SC,AP,nonce(Nsc)));
/* Step3 : */
MAPSays3 : MAP_Says3(ev,evs) <->
           EX AP, SC, Nsc, Nap :
           (NOT msgIN(nonce(Nap),used(evs)) AND
            eventIN(Says(AP,SC,pair(num(askRandom),
                         agent(AP))),evs) AND
            eventIN(Gets(AP,nonce(Nsc)),evs) AND
            ev = Says(AP,SC,crypt(msk(SC,AP,0),
                           pair(nonce(Nap),pair(nonce(Nsc),
                           pair(agent(SC),agent(AP))))));
/* Step4 : */
MAPSays4 : MAP_Says4(ev,evs) <->
           EX SC, AP, Nsc2, Nsc, Nap :
           (NOT msgIN(nonce(Nsc2),used(evs)) AND
            eventIN(Says(SC,AP,nonce(Nsc)),evs) AND
            eventIN(Gets(SC,crypt(msk(SC,AP,0),
                         pair(nonce(Nap),pair(nonce(Nsc),
                         pair(agent(SC),agent(AP))))),evs) AND
            ev = Says(SC,AP,crypt(msk(SC,AP,0),
                      pair(nonce(Nap),pair(nonce(Nsc2),agent(AP))))));
/* Step5 : */
MAPSays5 : MAP_Says5(ev,evs) <->
           EX AP, SC, Nap, Nsc, Nsc2 :
           (eventIN(Says(AP,SC,crypt(msk(SC,AP,0),
            pair(nonce(Nap),pair(nonce(Nsc),
            pair(agent(SC),agent(AP))))),evs) AND
            eventIN(Gets(AP,crypt(msk(SC,AP,0),
                         pair(nonce(Nap),
                         pair(nonce(Nsc2),agent(AP)))),evs) AND
            ev = Says(AP,SC,crypt(msk(SC,AP,0),
                      pair(num(getSessKey),nonce(Nsc2)))));
/* Step6 : */
MAPSays6 : MAP_Says6(ev,evs) <->
           EX SC, AP, Kapsc, Nap, Nsc2 :
           (NOT msgIN(key(Kapsc),used(evs)) AND
            sessKey(Kapsc) AND
```

```
              eventIN(Says(SC,AP,crypt(msk(SC,AP,0),
                           pair(nonce(Nap),pair(nonce(Nsc2),
                           agent(AP)))),evs) AND
              eventIN(Gets(SC,crypt(msk(SC,AP,0),
                           pair(num(getSessKey),nonce(Nsc2)))),evs) AND
              ev = Says(SC,AP,crypt(msk(SC,AP,0),
                        pair(nonce(Nsc2),key(Kapsc))));
  /* Oops1 : */
  MAPOops :  MAP_Oops(ev,evs) <->
             EX SC, AP, Nsc2, Kapsc :
             (eventIN(Says(SC,AP,crypt(msk(SC,AP,0),
                           pair(nonce(Nsc2),key(Kapsc)))),evs) AND
              sessKey(Kapsc) AND
              ev = Notes(spy,pair(nonce(Nsc2),key(Kapsc))))
 SATISFIES T_MAP_Properties
THEORYEND
```

B.2 Properties of the protocol in VSE-SL

```
THEORY T_MAP_Properties
 PURPOSE
 " Properties to be veriefied of the MAP protocol"
 USING T_MAP
 VARS ev : ProtocolEvent;
      evs : ProtocolTrace;
      A, B, AP, SC, AP2, SC2 : AgentT;
      n, Nsc22, Nsc2, Nap, Nsc : nat;
      Kapsc : KeyT;
      M, K : Msg;
      mL : MsgList
 AXIOMS
        /* Reg-Lem-1 */
        mskRegularityParts :
           (MAP(evs) AND
            msgIN(key(msk(A,B,n)),parts(gotInfo(spy,evs)))) ->
           (isBad(A) OR isBad(B));

        /* Reg-Lem-2 */
        mskRegularityAnalz :
           ALL evs, n, B, A :
           (MAP(evs) ->
           (msgIN(key(msk(A,B,n)),analz(gotInfo(spy,evs))) <->
           (isBad(A) OR isBad(B))));

        /* Authentication of AP by SC after receiving Msg3
           (Theorem AP-Auth-Thm) : Lem-3 */
        AP_Auth_Thm :
           (MAP(evs) AND
```

```
   eventIN(Says(SC,AP,nonce(Nsc)),evs) AND
   eventIN(Gets(SC,crypt(msk(SC,AP,0),
               pair(nonce(Nap),pair(nonce(Nsc),
               pair(agent(SC),agent(AP))))),evs) AND
   NOT isBad(SC) AND NOT isBad(AP)) ->
   eventIN(Says(AP,SC,crypt(msk(SC,AP,0),
               pair(nonce(Nap),pair(nonce(Nsc),
               pair(agent(SC),agent(AP))))),evs);

/* Authenticity of Msg3 from the point of view of SC
   (Lemma AP-Auth-Lem) : Lem-4 */
AP_Auth_Lem :
   (MAP(evs) AND
    msgIN(crypt(msk(SC,AP,0),pair(nonce(Nap),
          pair(nonce(Nsc),pair(agent(SC),agent(AP))))),
          parts(gotInfo(spy,evs))) AND
    NOT isBad(SC) AND NOT isBad(AP)) ->
    eventIN(Says(AP,SC,crypt(msk(SC,AP,0),
                pair(nonce(Nap),pair(nonce(Nsc),
                pair(agent(SC),agent(AP))))),evs);

/* Authentication of SC by AP after receiving Msg4
   (Theorem SC-Auth-Thm) : Lem-5 */
SC_Auth_Thm :
   (MAP(evs) AND
    eventIN(Says(AP,SC,crypt(msk(SC,AP,0),
                pair(nonce(Nap),pair(nonce(Nsc),
                pair(agent(SC),agent(AP))))),evs) AND
    eventIN(Gets(AP,crypt(msk(SC,AP,0),
                pair(nonce(Nap),
                pair(nonce(Nsc2),agent(AP))))),evs) AND
    NOT isBad(SC) AND NOT isBad(AP)) ->
    eventIN(Says(SC,AP,crypt(msk(SC,AP,0),
                pair(nonce(Nap),
                pair(nonce(Nsc2),agent(AP))))),evs);

/* Authenticity of Msg4 from the point of view of AP
   (Lemma SC-Auth-Lem) : Lem-6  */
SC_Auth_Lem :
   (MAP(evs) AND
    msgIN(crypt(msk(SC,AP,0),pair(nonce(Nap),
          pair(nonce(Nsc2),agent(AP)))),
          parts(gotInfo(spy,evs))) AND
    NOT isBad(SC) AND NOT isBad(AP)) ->
    eventIN(Says(SC,AP,crypt(msk(SC,AP,0),pair(nonce(Nap),
                pair(nonce(Nsc2),agent(AP))))),evs);

/* Confidentiality property of the session key
   (Theorem Key-Conf-Thm) : Lem-7 */
```

```
        Key_Conf_Thm :
           (MAP(evs) AND
           eventIN(Says(SC,AP,crypt(msk(SC,AP,0),
                       pair(nonce(Nsc2),key(Kapsc)))),evs) AND
           NOT isBad(SC) AND NOT isBad(AP) AND
           NOT eventIN(Notes(spy,pair(nonce(Nsc2),key(Kapsc))),evs)) ->
           NOT msgIN(key(Kapsc),analz(gotInfo(spy,evs)));

        /* Authenticity of Msg6 from the point of view of AP
           (Theorem Key-Auth-Thm) : Lem-8 */
        Key_Auth_Thm :
           (MAP(evs) AND
            msgIN(crypt(msk(SC,AP,0),pair(nonce(Nap),
                  pair(nonce(Nsc2),agent(AP)))),
                  parts(gotInfo(spy,evs))) AND
            msgIN(crypt(msk(SC,AP,0),pair(nonce(Nsc2),
                  ey(Kapsc))),parts(gotInfo(spy,evs))) AND
            NOT isBad(SC) AND NOT isBad(AP)) ->
            eventIN(Says(SC,AP,crypt(msk(SC,AP,0),pair(nonce(Nsc2),
                         key(Kapsc)))),evs);

        /* Unicity of the session key in Msg6
           (Theorem Key-Unicity-Thm) : Lem-9 */
        Key_Unicity_Thm :
           (MAP(evs) AND
            NOT msgIN(key(Kapsc),analz(gotInfo(spy,evs)))) ->
            EX SC2, AP2, Nsc22 : ALL SC, AP, Nsc2 :
            (eventIN(Says(SC,AP,crypt(msk(SC,AP,0),
                          pair(nonce(Nsc2),key(Kapsc)))),evs) ->
            (SC = SC2 AND AP = AP2 AND Nsc2 = Nsc22))
THEORYEND
```

Appendix C

VSE model of the Extended SSCM

C.1 Defining system states in VSE-SL

```
1. Bnet
=======

BASIC Bnet
 USING T_MAP
 netentry = mknetentry(sender : nat,receiver : nat,entry : Msg)
            WITH isnetentry
BASICEND

2. BnetList
===========

BASIC BnetList
 USING Bnet
 Netentrylist = NullNetentry WITH isNullNetentry |
                addnetentry(firstnetentry : netentry,
                            restnetentry : Netentrylist)
                WITH isAddNetEntry
BASICEND


3. Bstate
=========

BASIC Bstate
 USING Keywords
 stateentry = stateInfo(stateInfo_arg : StateLabel)
              WITH isstateInfoentry |
```

```
              storeInfo(storeInfo_arg : StoreItem)
              WITH isstoreInfoentry
BASICEND


4. Bstatelist
=============

BASIC Bstatelist
 USING Bstate
 stateentrylist = nullentry WITH isnullentry |
                  addstateentry(firststateentry : stateentry,
                                reststateentry : stateentrylist)
                  WITH isaddstateentry
BASICEND


5. Keywords
===========

THEORY Keywords
 USING T_MAP
 TYPES Component =
       FREELY GENERATED BY observer |
                           participant;
       StateLabel =
       FREELY GENERATED BY
                     start |
                     sent1(sent1_arg : nat) |
                     rcvd2(rcvd2_arg1 : nat,rcvd2_arg2 : Msg) |
                     sent3(sent3_arg1 : nat,sent3_arg2 : Msg,
                           sent3_arg3 : Msg) |
                     rcvd4(rcvd4_arg1 : nat,rcvd4_arg2 : Msg) |
                     sent5(sent5_arg1 : nat,sent5_arg2 : Msg) |
                     rcvd6(rcvd6_arg1 : nat,rcvd6_arg2 : Msg) |
                     ready |
                     rcvd1(rcvd1_arg : nat) |
                     sent2(sent2_arg1 : nat,sent2_arg2 : Msg) |
                     rcvd3(rcvd3_arg1 : nat,rcvd3_arg2 : Msg,
                           rcvd3_arg3 : Msg) |
                     sent4(sent4_arg1 : nat,sent4_arg2 : Msg) |
                     rcvd5(rcvd5_arg1 : nat,rcvd5_arg2 : Msg) |
                     sent6(sent6_arg1 : nat,sent6_arg2 : Msg);
       StoreItem =
       FREELY GENERATED BY
                     authKey(authKey_arg1 : nat,authKey_arg2 : KeyT) |
                     encKey(encKey_arg1 : nat,encKey_arg2 : KeyT)
THEORYEND
```

```
6. localState
=============

THEORY localState
 USING Bstatelist;
       arrays[Bstatelist.stateentrylist]
 FUNCTIONS delstateentry : nat, stateentry, array -> array;
           enterstateentry : nat, stateentry, array -> array;
           max : nat;
           nth_sesskey : nat -> KeyT
 PREDICATES inState : stateentry, stateentrylist
 VARS n, m : nat
 AXIOMS
 sessKey(nth_sesskey(n));
 nth_sesskey(n) = nth_sesskey(m) -> n = m
THEORYEND


7. netState
===========

THEORY netState
 USING BnetList
 FUNCTIONS deletenetentry : netentry, netentrylist -> netentrylist
 PREDICATES netmember : netentry, netentrylist
 VARS entry, entry1 : netentry;
      rst : Netentrylist
 AXIOMS
 deletenetentry(entry,NullNetentry) = NullNetentry;
 deletenetentry(entry,addnetentry(entry,rst)) = deletenetentry(entry,rst);
 entry /= entry1 -> deletenetentry(entry,addnetentry(entry1,rst)) =
                    addnetentry(entry1,deletenetentry(entry,rst));
 NOT netmember(entry,NullNetentry);
 netmember(entry,addnetentry(entry1,rst)) <->
 (entry = entry1 OR netmember(entry,rst))
THEORYEND


8. systemState
==============

THEORY systemState
 USING netState;
       localState;
       BOOLEAN
THEORYEND
```

C.2 VSE-SL Specification of the eSSCM

```
1. Combined_SSCM
================

TLSPEC Combined_SSCM
 PURPOSE
 "Defines all connections of the components and the entire system "
 USING Definition
 DATA OUT h1_put : list
      OUT h2_scout : list
      OUT h3_appl : list
      OUT trace : ProtocolTrace
      OUT StateSC : array
      OUT StateAP : array
      OUT Obsnet : netentrylist
 VARS m1, m2, m3, m4, m5 : msg;
      SC, AP, X, idx, idx2 : nat;
      shK : KeyT;
      kw, kw1, kw2 : StateLabel
 COMBINE KeyGen [KeyGen.h1_put -> Combined_SSCM.h1_put] ;
         CommC [CommC.sc_in <- ExtAppl.sc_in,
                CommC.h2_scout -> Combined_SSCM.h2_scout,
                CommC.StateSC -> Combined_SSCM.StateSC]
               SHARED [CommC.key_list <- KeyGen.key_list];
         ExtAppl [ExtAppl.h3_appl -> Combined_SSCM.h3_appl,
                  ExtAppl.sc_out <- CommC.sc_out,
                  ExtAppl.StateAP -> Combined_SSCM.StateAP] ;
         Observer [Observer.trace -> Combined_SSCM.trace,
                   Observer.obsnet -> Combined_SSCM.obsnet]
 SATISFIES SSCM_Security_Model
TLSPECEND

2. CommC
========

TLSPEC CommC
 PURPOSE
 " Communication Component of the Smart Card "
 USING Definition
 DATA
      /* channel get from buffer */
      SHARED INOUT key_list : list
      /* schedule varibale for observer */
      SHARED INOUT next : Component
      /* global variable for net state for observer */
      SHARED INOUT net : Netentrylist
      /* channel sc_in from ExtAppl */
      IN sc_in : Msg
```

```
     /* channel sc_out to ExtAppl */
     OUT sc_out : Msg
     /* history variable of channel sc_out */
     OUT h2_scout : list
     /* local trace of internal states of SC */
     OUT StateSC : array
     INTERNAL Rsp, Chlng : Msg;
              k_apsc, n_sc : nat
VARS Nsc, Kapsc, Nap : Msg;
     SC, AP, X, idx, idx1, idx2 : nat;
     shK : KeyT;
     kw, kw1 : StateLabel
ACTIONS

/* step1 of the map protocol -> SC receives (gets) the
   msg "askRandom" from AP */
receive1 ::=
   IF next = participant AND
      sc_in = pair(num(askRandom), agent(friend(AP)))
   THEN EX SC, AP, X :
      (SC <= max AND
       inState(stateInfo(ready), select(StateSC, SC)) AND
       netmember(mknetentry(X, SC, pair(num(askRandom),
                            agent(friend(AP)))), net) AND
       AP <= max AND
       StateSC' = enterstateentry(SC, stateInfo(rcvd1(AP)),
                  delstateentry(SC, stateInfo(ready), StateSC)) AND
       net' = deletenetentry(mknetentry(X, SC, pair(num(askRandom),
                agent(friend(AP)))), net)) AND
       next' = observer AND
       UNCHANGED(key_list, h2_scout, sc_out, Rsp, Chlng, n_sc, k_apsc)
   ELSE UNCHANGED(key_list, next, net, sc_out, h2_scout,
                  StateSC, Rsp, Chlng, n_sc, k_apsc)
   FI

/* step2 of the map protocol -> SC sends (says) the nonce Nsc */
send2 ::=
   IF next = participant AND
      key_list /= nil
   THEN EX SC, AP :
      (SC <= max AND
       inState(stateInfo(rcvd1(AP)), select(StateSC, SC)) AND
       n_sc' = last(key_list) AND
       Chlng' = nonce(n_sc') AND
       key_list' = butlast(key_list) AND
       sc_out' = pair(agent(friend(AP)), nonce(n_sc')) AND
       h2_scout' = cons(n_sc', h2_scout) AND
       StateSC' = enterstateentry(SC, stateInfo(sent2(AP, Nsc)),
                  delstateentry(SC, stateInfo(rcvd1(SC)), StateSC)) AND
```

```
       net' = Bnetlist.addnetentry(mknetentry(SC, AP, Nsc), net)) AND
       next' = observer AND
       UNCHANGED(Rsp, k_apsc)
   ELSE UNCHANGED(key_list, next, net, sc_out, h2_scout, StateSC,
                  Rsp, Chlng, n_sc, k_apsc)
   FI

/* step3 of the map protocol -> SC receives (gets) the cipher from AP */
receive3 ::=
   IF next = participant AND
      sc_in = crypt(shK, pair(Nap, pair(Nsc,pair(agent(friend(SC)),
                              agent(friend(AP))))))
   THEN EX SC, AP, Nsc, shK, X, Nap :
      (SC <= max AND
       inState(stateInfo(sent2(AP, Nsc)), select(StateSC, SC)) AND
       inState(storeInfo(authKey(AP, shK)), select(StateSC, SC)) AND
       netmember(mknetentry(X, SC, crypt(shK, pair(Nap,
                 pair(Nsc, pair(agent(friend(SC)),
                 agent(friend(AP)))))), net) AND
       Rsp' = Nsc AND
       StateSC' = enterstateentry(SC, stateInfo(rcvd3(AP, Nsc, Nap)),
                  delstateentry(SC,stateInfo(sent2(AP,Nsc)),StateSC)) AND
       net' = deletenetentry(mknetentry(X, SC, crypt(shK,
                            pair(Nap, pair(Nsc, pair(agent(friend(SC)),
                                 agent(friend(AP)))))), net)) AND
       next' = observer AND
       UNCHANGED(key_list, sc_out, h2_scout, Chlng, n_sc, k_apsc)
   ELSE UNCHANGED(key_list, next, net, sc_out, h2_scout,
                  StateSC, Rsp, Chlng, n_sc, k_apsc)
   FI

/* step4 of the map protocol -> SC sends (says) the cipher to AP */
send4 ::=
   IF next = participant AND
      key_list /= nil AND
      Chlng = Rsp
   THEN EX SC, AP, Nsc, Nap, shK :
      (SC <= max AND
       inState(stateInfo(rcvd3(AP, Nsc, Nap)), select(StateSC, SC)) AND
       inState(storeInfo(authKey(AP, shK)), select(StateSC, SC)) AND
       n_sc' = last(key_list) AND
       Chlng' = nonce(n_sc') AND
       key_list' = butlast(key_list) AND
       sc_out' = crypt(shK, pair(Nap, pair(nonce(n_sc'),
                                 agent(friend(AP))))) AND
       h2_scout' = cons(n_sc', h2_scout) AND
       StateSC' = enterstateentry(SC, stateInfo(sent4(AP, Nsc)),
                  delstateentry(SC, stateInfo(rcvd3(AP, Nsc, Nap)),
                                          StateSC)) AND
```

```
      net' = addnetentry(mknetentry(SC, AP, crypt(shK, pair(Nap,
                        pair(nonce(n_sc'),agent(friend(AP)))))), net))
                        AND
      next' = observer AND
      UNCHANGED(Rsp, k_apsc)
   ELSE UNCHANGED(key_list, next, net, sc_out, h2_scout, StateSC, Rsp,
                  Chlng, n_sc, k_apsc)
   FI

/* step5 of the map protocol -> SC receives (gets)
   getSessionKey from AP */
receive5 ::=
   IF next = participant AND
      sc_in = crypt(shK, pair(num(getSessKey), Nsc))
   THEN EX SC, AP, Nsc, shK, X :
      (SC <= max AND
       inState(stateInfo(sent4(AP, Nsc)), select(StateSC, SC)) AND
       netmember(mknetentry(X, SC, crypt(shK, pair(num(getSessKey),
                            Nsc))), net) AND
       Rsp' = Nsc AND
       StateSC' = enterstateentry(SC, stateInfo(rcvd5(AP, Nsc)),
                  delstateentry(SC, stateInfo(sent4(SC, Nsc)), StateSC))
                  AND
       net' = deletenetentry(mknetentry(X, SC, crypt(shK,
                             pair(num(getSessKey), Nsc))), net)) AND
       next' = observer AND
   UNCHANGED(key_list, sc_out, h2_scout, Chlng, n_sc, k_apsc)
   ELSE UNCHANGED(key_list, next, net, sc_out, h2_scout, StateSC, Rsp,
                  Chlng, n_sc, k_apsc)
   FI

/* step6 of the map protocol -> SC sends (says) sessionKey to AP */
send6 ::=
   IF next = participant AND
      Chlng = Rsp AND
      key_list /= nil
   THEN EX SC, AP, Nsc, shK :
      (SC <= max AND
       Chlng = Nsc AND
       inState(stateInfo(rcvd5(AP, Nsc)), select(StateSC, SC)) AND
       inState(storeInfo(encKey(AP, shK)), select(StateSC, SC)) AND
       k_apsc' = last(key_list) AND
       key_list' = butlast(key_list) AND
       sc_out' = crypt(shK, pair(Chlng, nonce(k_apsc'))) AND
       h2_scout' = cons(trans(k_apsc'), h2_scout) AND
       StateSC' = enterstateentry(SC, stateInfo(ready),
                  delstateentry(SC, stateInfo(rcvd5(AP, Nsc)), StateSC))
                  AND
       net' = addnetentry(mknetentry(SC, AP, crypt(shK, pair(Nsc,
```

```
                                        Kapsc))), net)) AND
       next' = observer AND
       UNCHANGED(Chlng, Rsp)
   ELSE UNCHANGED(key_list, next, net, sc_out, h2_scout, StateSC, Rsp,
                  Chlng, n_sc, k_apsc)
   FI

 SPEC
     /* Specification of the behavior of component CommC */
     INITIAL key_list = nil AND
             h2_scout = nil AND
             (idx <= max ->
              ((inState(stateInfo(kw), select(StateSC, idx)) ->
                kw = ready) AND
               ALL idx2 :
               idx2 <= max ->
               inState(storeInfo(authKey(idx1, msk(friend(idx),
                       friend(idx1), 0))), select(StateSC, idx)) ->
               idx1 = idx2 AND
               inState(storeInfo(encKey(idx1, msk(friend(idx),
                       friend(idx1), succ(0)))), select(StateSC, idx)) ->
               idx1 = idx2))
     TRANSITIONS [receive1, send2, receive3, send4, receive5, send6]
                 {key_list, StateSC, Chlng, Rsp, sc_out, h2_scout,
                  n_sc, k_apsc}
 HIDE Chlng, Rsp, n_sc, k_apsc
TLSPECEND

3. Definition
=============

THEORY Definition
 USING Channel_Data_LQs;
       systemState
 FUNCTIONS inv_trans : nat -> nat;
           trans : nat -> nat;
           Msg2nat : Msg -> nat
 PREDICATES v_check : nat;
            p_check : nat;
            p_check_list : list;
            greatest_elem : nat, list
 VARS k, e : nat;
      l : list
 AXIOMS FOR trans : p_check(k) ->
                    NOT p_check(trans(k))
        FOR p_check_list : p_check_list(l) <->
                           l = nil OR
                           (p_check(first(l)) AND
                            p_check_list(rest(l)))
```

```
        FOR p_check_list : p_check_list(l) ->
                           p_check_list(butlast(l))
        FOR p_check_list : (p_check_list(l) ->
                            p_check(last(l)))
        FOR inv_trans : k = inv_trans(trans(k))
        FOR greatest_elem : (greatest_elem(k,l) <->
                             l = nil OR
                             (k > first(l) AND
                              greatest_elem(k,rest(l))))
THEORYEND

4. ExtAppl
==========

TLSPEC ExtAppl
 PURPOSE
 " External Application receiving a key from CommC "
 USING Definition
 DATA
      /* schedule varibale for observer */
      SHARED INOUT next : Component
      /* global variable for net state for observer */
      SHARED INOUT net : Netentrylist
      INTERNAL
               /* holding received session key */
               xc, xn : nat;
               Chlng, Nonce, Rsp : Msg;
               /* holds nonces internally generated */
               nonces_list : list
      OUT
          /* send key request and authentication data to CommC
             via channel sc_in */
          sc_in : Msg;
          /* local trace of internal state of AP */
          StateAP : array
      /* history variable of received keys */
      OUT h3_appl : list
      /* receive data from channel sc_out */
      IN sc_out : Msg
 VARS Nsc, Nsc2, Kapsc, Nap : Msg;
      AP, SC, X, idx, idx2, k_apsc, n_ap : nat;
      shK : KeyT
 ACTIONS
 /* step1 of the map protocol -> AP sends (says) askRandom to SC */
 send1 ::=
     next = participant AND
     EX AP, SC :
     (AP <= max AND
      inState(stateInfo(start), select(StateAP, AP)) AND
```

```
     SC <= max AND
     AP /= SC AND
     StateAP' = enterstateentry(AP, stateInfo(sent1(SC)),
                delstateentry(AP, stateInfo(start), StateAP)) AND
     sc_in' = pair(num(askRandom), agent(friend(AP))) AND
     net' = addnetentry(mknetentry(AP, SC, pair(num(askRandom),
                       agent(friend(AP)))), net)) AND
     next' = observer AND
     UNCHANGED(h3_appl, xc, Chlng, Rsp, xn, Nonce)

/* step2 of the map protocol -> AP receives (gets) nonce Nsc from SC */
receive2 ::=
    next = participant AND
    EX AP, SC, X, Nsc :
    (AP <= max AND
     inState(stateInfo(sent1(SC)), select(StateAP, AP)) AND
     netmember(mknetentry(X, AP, Nsc), net) AND
     sc_out = pair(agent(friend(AP)), Nsc) AND
     StateAP' = enterstateentry(AP, stateInfo(rcvd2(AP, Nsc)),
                delstateentry(AP, stateInfo(sent1(SC)), StateAP)) AND
     net' = deletenetentry(mknetentry(X, AP, Nsc), net)) AND
     next' = observer AND
     UNCHANGED(h3_appl, xc, Chlng, Rsp, xn, Nonce)

/* step3 of the map protocol -> AP sends (says) cipher to SC */
send3 ::=
   next = participant AND
   EX AP, SC, Nsc, shK :
   (AP <= max AND
    inState(stateInfo(rcvd2(SC, Nsc)), select(StateAP, AP)) AND
    inState(storeInfo(authKey(SC, shK)), select(StateAP, AP)) AND
    nonces_list /= nil AND
    n_ap = last(nonces_list) AND
    Nap = nonce(n_ap) AND
    Chlng' = Nap AND
    nonces_list' = butlast(nonces_list) AND
    sc_in' = crypt(shK, pair(Nap, pair(Nsc, pair(agent(friend(SC)),
                  agent(friend(AP)))))) AND
    StateAP' = enterstateentry(AP, stateInfo(sent3(SC, Nsc, Nap)),
               delstateentry(AP, stateInfo(rcvd2(SC, Nsc)), StateAP)) AND
    net' = addnetentry(mknetentry(AP, SC, crypt(shK, pair(Nap, pair(Nsc,
           pair(agent(friend(SC)), agent(friend(AP))))))), net)) AND
    next' = observer AND
    UNCHANGED(h3_appl, Rsp, xn, Nonce)

/* step4 of map protocol -> AP receives (gets) cipher from SC */
receive4 ::=
    next = participant AND
    EX AP, SC, Nsc, Nap, shK, X, Nsc2 :
```

```
    (AP <= max AND
     inState(stateInfo(sent3(SC, Nsc, Nap)), select(StateAP, AP)) AND
     inState(storeInfo(authKey(SC, shK)), select(StateAP, AP)) AND
     netmember(mknetentry(X, AP, crypt(shK, pair(Nap, pair(Nsc2,
                          agent(friend(AP)))))), net) AND
     sc_out = crypt(shK, pair(Nap, pair(Nsc2, agent(friend(AP))))) AND
     Rsp' = Nap AND
     Nonce' = Nsc2 AND
     StateAP' = enterstateentry(AP, stateInfo(rcvd4(SC, Nsc)),
                delstateentry(AP, stateInfo(sent3(SC, Nsc, Nap)),
                              StateAP)) AND
     net' = deletenetentry(mknetentry(X, AP, crypt(shK, pair(Nap,
                           pair(Nsc2, agent(friend(AP)))))), net)) AND
     next' = observer AND
     UNCHANGED(h3_appl, xn, Chlng)

/* step5 of the map protocol -> AP sends (says) getSessionKey to SC */
send5 ::=
    next = participant AND
    EX AP, SC, Nsc :
    (AP <= max AND
     inState(stateInfo(rcvd4(SC, Nsc)), select(StateAP, AP)) AND
     Chlng = Rsp AND
     Nsc = Nonce AND
     sc_in' = crypt(shK, pair(num(getSessKey), Nsc)) AND
     StateAP' = enterstateentry(AP, stateInfo(sent5(SC, Nsc)),
                delstateentry(AP, stateInfo(rcvd4(SC, Nsc)), StateAP)) AND
     net' = addnetentry(mknetentry(AP, SC, crypt(shK,
                        pair(num(getSessKey), Nsc))), net)) AND
     next' = observer AND
     UNCHANGED(h3_appl, xc, Nonce, Chlng, xn, Rsp)

/* step6 of the map protocol -> AP receives (gets) the
   sessionKey from SC */
receive6 ::=
   next = participant AND
   EX AP, SC, Nsc, shK, X, Kapsc :
   (AP <= max AND
    inState(stateInfo(sent5(SC, Nsc)), select(StateAP, AP)) AND
    inState(storeInfo(encKey(SC, shK)), select(StateAP, AP)) AND
    netmember(mknetentry(X, AP, crypt(shK, pair(Nsc, Kapsc))), net) AND
    sc_out = crypt(shK, pair(Nsc, Kapsc)) AND
    Nonce = Nsc AND
    k_apsc = Msg2nat(Kapsc) AND
    h3_appl' = cons(inv_trans(k_apsc), h3_appl) AND
    StateAP' = enterstateentry(AP, stateInfo(start), delstateentry(AP,
                               stateInfo(sent5(SC, Nsc)), StateAP)) AND
    net' = deletenetentry(mknetentry(X, AP, crypt(shK, pair(Nsc,
                                    Kapsc))), net)) AND
```

```
    next' = observer AND
    UNCHANGED(xc, Chlng, Rsp, xn, Nonce)

 /* action generates nonces and stores them in the internal list */
 generate ::= xn' = xn + 1 AND
              nonces_list' = cons(xn', nonces_list) AND
              UNCHANGED(xc, Chlng, Rsp, Nonce, sc_in, h3_appl, StateAP)

 SPEC INITIAL h3_appl = nil AND
              nonces_list = nil AND
              next = participant AND
              ALL idx :
              (idx <= max ->
               inState(stateInfo(start), select(StateAP, idx)) AND
               ALL idx2 :
               (idx2 <= max ->
                (inState(storeInfo(authKey(idx2, msk(friend(idx),
                 friend(idx2), 0))), select(StateAP, idx)) AND
                 inState(storeInfo(encKey(idx2, msk(friend(idx),
                 friend(idx2), succ(0)))), select(StateAP, idx)))))
     TRANSITIONS [send1, receive2, send3, receive4, send5, receive6,
                  generate]{xc, sc_in, StateAP, h3_appl, Chlng, Rsp,
                  xn, Nonce}
 HIDE xc, Chlng, Rsp, xn, nonces_list, Nonce
TLSPECEND

5. KeyGen
=========

TLSPEC KeyGen
 PURPOSE
 " Generation of keys "
 USING Definition
 DATA
      /* channel put to buffer */
      SHARED INOUT key_list : list
      /* history variable of channel put */
      OUT h1_put : list
      /* internal computed random value */
      INTERNAL xp : nat
 ACTIONS
 /* Specification of the compute action.
    The computation is left unspecified. */
 generate ::= xp' = xp + 1;
              UNCHANGED(key_list, h1_put)
 /* specification of the predicate and the send function */
 check_send ::= IF p_check(xp)
                THEN key_list' = cons(xp, key_list) AND
                     h1_put' = cons(xp, h1_put) AND
```

```
                    UNCHANGED(xp)
                ELSE UNCHANGED(xp, key_list, h1_put)
                FI
 SPEC
      /* Specification of the behavior of the producer */
      INITIAL BEGIN
               xp := 1;
               h1_put := nil;
               key_list := nil
              END
      TRANSITIONS BEGIN
                   WHILE TRUE DO
                    generate;
                    check_send
                   OD
                  END {key_list, h1_put, xp}
 /* Variables not externally visible */
 HIDE xp
TLSPECEND

6. Observer
===========

TLSPEC Observer
 USING Definition
 DATA
      /* Scheduling */
      SHARED INOUT next : Component
      /* Messages from the agents at work */
      OUT trace : ProtocolTrace
      /*Observations of the net to detect says- and gets-
          events:
          Obsnet = net U {(A,B,M)} ==> gets(friend(B),M)
          Obsnet = net \ {(A,B,M)} ==> says(friend(A),friend(B),M)
          */
      SHARED INOUT net : Netentrylist
      /*  detecting changes.of "net" */
      OUT Obsnet : Netentrylist
 VARS X, Y, X1, Y1 : nat;
      newEv : ProtocolEvent;
      evtP : Bool;
      M, M1 : Msg
 ACTIONS
 buildtrace ::=
    next = observer AND
    next' = participant AND
    ((Obsnet = net ->
      evtP = F) AND
     (Obsnet /= net ->
```

```
      (evtP = T AND
       (EX X, Y, M :
        (net = Bnetlist.addnetentry(mknetentry(X, Y, M), Obsnet) AND
         newEv = says(friend(X), friend(Y), M)) OR
        (net = deletenetentry(mknetentry(X, Y, M), Obsnet) AND
         newEv = gets(friend(Y), M)))))) AND
    (evtP = T ->
     trace' = addEvent(newEv, trace)) AND
    (evtP = F ->
     trace' = trace) AND
    Obsnet' = net
 SPEC INITIAL trace = nullEvent
      TRANSITIONS [buildtrace] {trace, Obsnet}
TLSPECEND


7. SSCM_Security_Model
======================

TLSPEC SSCM_Security_Model
 PURPOSE
 " Specification of the security model of the overall system "
 USING Definition
 INCLUDE S_S_C_M = Combined_SSCM
 SPEC
      /* confidentiality property:
           all transmitted keys from CommC to ExtAppl are
           readable for the authenticated ExtAppl only, or
           a received key can never occur in the delivery channel
          */
      [] (S_S_C_M.h2_scout /= nil ->
          NOT p_check(first(S_S_C_M.h2_scout)));
      /* integrity property:
           any delivered key has not been changed
           */
      [] (S_S_C_M.h3_appl /= nil ->
          p_check(first(S_S_C_M.h3_appl)));
      /* non-duplicating property:
           any delivered key can not appear twice in consumer's
           output list (or in terms of the model: the first element
           in the list is always the greatest element of the list) */
      [] (S_S_C_M.h3_appl /= nil ->
          greatest_elem(first(S_S_C_M.h3_appl), rest(S_S_C_M.h3_appl)));
      /* protocol property: a trace given by the observer should
         always be valid map trace */
      [] MAP(S_S_C_M.trace)
TLSPECEND
```

Bibliography

[ACL03] Andronick, J.; Chatali, B.; Ly, O.: Using Coq to verify Java Card applet isolation properties. In: *16th International Conference on Theorem Proving in Higher Order Logics*, no. 2758 in LNCS. Springer, 2003.

[AL91] Abadi, M.; Lamport, L.: The existence of refinement mappings. In: *Theoretical Computer Science*, volume 82(2):pp. 253–284, 1991.

[AM96] Abadi, Martín; Merz, Stephan: On TLA as a logic. In: Broy, Manfred, editor, *Deductive Program Design*, NATO ASI series F, pp. 235–272. Springer-Verlag, Berlin, 1996.

[And01] Anderson, R.J.: *Security Engineering - A guide to building dependable distributed systems*. John Wiley & Sons, 2001.

[BAN89] Burrows, Michael; Abadi, Martín; Needham, Roger: A Logic of Authentication. In: *Proceedings of the Royal Society*, Volume 426, Number 1871. 1989.

[Bel00] Bella, G.: *Inductive Verification of Cryptographic Protocols*. PhD thesis, Cambridge University, 2000.

[Bel07] Bella, G.: *Formal Correctness of Security Protocols*. Springer-Verlag's Information Security and Cryptography series, 2007.

[BHHW86] Biundo, S.; Hummel, B.; Hutter, D.; Walther, C.: The Karlsruhe Induction Theorem Proving System. In: *Jörg Siekmann, editor,8th Conference on Automated Deduction*, p. 672. 1986.

[Bib77] Biba, K.J.: Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, MITRE Corporation, 1977.

[Bis03] Bishop, M: *Computer Security: Art and Science*. Addison-Wesley, 2003.

[BL76] Bell, D.E.; LaPadula, L.: Secure Computer Systems: Unified Exposition and Multics Interpretation. Technical Report MTR-2997, MITRE Corporation, 1976.

[BM79] Boyer, R.S.; Moore, J.S.: *A Computational Logic*. Academic Press, New York, 1979.

[BN89] Brewer, D. F. C.; Nash, M. J.: The Chinese Wall Security Policy. In: *IEEE Symposium on Security and Privacy*, pp. 206–214. 1989.

[BP06] Bella, Giampaolo; Paulson, Lawrence C.: Accountability protocols: Formalized and verified. In: *ACM Trans. Inf. Syst. Secur.*, volume 9(2):pp. 138–161, 2006.

[BPW03] Backes, M.; Pfitzmann, B.; Waidner, M.: A composable cryptographic library with nested operations. In: *Proc. 10th ACM Conference on Computer and Communications Security*, pp. 220–230. 2003.

[BSI89] Criteria for the Evaluation of Trustworthiness of Information Technology (IT) Systems. Federal Office for Information Security (BSI), Germany, 1989.

[BSI00] VSE/BSI-Project - Specification Language VSE-SL. 2000. Federal Office for Information Security (BSI), Germany.

[BSI04a] Common Criteria Protection Profile for Biometric Verification Mechanisms. 2004. Federal Office for Information Security (BSI), Germany.

[BSI04b] Guideline for the Development and Evaluation of Formal Security Policy Models in the scope of ITSEC and Common Criteria. December 2004. Federal Office for Information Security (BSI), Germany.

[CC99] Information Technology - Security Techniques - Evaluation Criteria for IT Security - Part 1: Introduction and General Model. 1999. International Organization for Standardization, ISO/IEC 15408-1.

[CC02] Biometric Evaluation Methodology Supplement. 2002. Common Criteria - Common Methodology for Informatin Technology Security Evaluation.

[CES89] UK Systems Security Confidence Levels, CESG Memorandum No. 3. 1989. Communications-Electronics Security Group, United Kingdom.

[CRS+06] Cheikhrouhou, L.; Rock, G.; Stephan, W.; Schwan, M.; Lassmann, G.: Verifying a Chipcard-Based Biometric Identification Protocol in VSE. In: Janusz Górski, editor, *Computer Safety, Reliability, and Security, 25th International Conference, SAFECOMP 2006*, volume 4166 of *LNCS*, pp. 42–56. Springer, 2006.

[CSI85] Canadian Trusted Products Evaluation Criteria (CTPEC). 1985. Communications Security Establishment (CSE), Canada.

[CW87] Clark, D. D.; Wilson, D. R.: A Comparison of Commercial and Military Computer Security Policies. In: *IEEE Symposium on Security and Privacy*, pp. 184–195. 1987.

[Den76] Denning, Dorothy E.: A lattice model of secure information flow. In: *Commun. ACM*, volume 19(5):pp. 236–243, 1976.

[DIR99] DIRECTIVE 1999/93/EC OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 13 December 1999 on a Community Framework for Electronic Signatures. 1999. Official Journal L 013 , 19/01/2000 p. 0012 - 0020.

[DoD85] Trusted Computer System Evaluation Criteria (TCSEC). 1985. Department of Defense of the U.S., DoD 5200.28-STD.

[DTI89] DTI Commercial Computer Security Centre Evaluation Levels Manual. 1989. Department of Trade and Industry, United Kingdom.

[DY81] Dolev, Danny; Yao, A.C.: On the security of public key protocols. Technical report, Stanford University, Stanford, CA, USA, 1981.

[Eck04] Eckert, C.: *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. Oldenburg Wissenschaftsverlag, 3rd edition, 2004.

[FIP01] Federal Information Processing Standard (FIPS) 140-2. May 2001. Institute of Computer Sciences and Technology (ICST) in the U.S.

[FK92] Ferraiolo, D.F.; Kuhn, D.R.: Role Based Access Control. In: *Proceedings of the 15. NIST National Computer Security Conference*, pp. 554–563. 1992.

[Gir99] Girard, P.: Which Security Policy for Multiapplication Smart Card. In: *USENIX Workshop on Smart Card Technology*, pp. 21–28. Usenix Association, 1999.

[GM82] Goguen, J.; Meseguer, J.: Security policies and security models. In: *IEEE Symposium on Research in Security and Privacy*, pp. 11–20. IEEE Computer Society Press, 1982.

[GM84] Goguen, Joseph A.; Meseguer, José: Unwinding and inference control. In: *IEEE Symposium on Security and Privacy*, pp. 75–87. 1984.

[GNY90] Gong, L.; Needham, R.; Yahalom, R.: Reasoning about belief in cryptographic protocols. In: *Proceeding of the Symposium on Research in Security and Privacy volume 4/90*, pp. 234–248. IEEE Computer Society Press, 1990.

[Göd31] Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. In: *Monatshefte für Mathematik und Physik*, volume 38:pp. 173–198, 1931.

[Gol97] Goldstein, T.: The gateway security model in the Java electronic commerce framework. In: *Proceedings of the Financial Cryptography '97*. Springer, 1997.

[Gol99] Gollmann, D.: *Computer Security*. John Wiley & Sons, 1999.

[GS06] Giessmann, E.G.; Schwan, M.: Formales Sicherheitsmodell des TC-Schlüsselgenerators 2.0. Technical report, Produktzentrum T-TeleSec der Deutschen Telekom AG, Siegen, 2006. Produktzentrum T-TeleSec der Deutschen Telekom AG, Evaluationsunterlagen des TÜViT, Verschlußsache (Confidential).

[HLsS+96] Hutter, D.; Langenstein, B.; Sengler, C.; Siekmann, J.H.; Stephan, W.; Wolpers, A.: Deduction in the Verification Support Environment (VSE). In: *Proceedings Formal Methods Europe 1996*, Industrial Benefits and Advances in Formal Methods. Springer-Verlag, Berlin, 1996.

[HMR+99] Hutter, D.; Mantel, H.; Rock, G.; Stephan, W.; Wolpers, A.; Balser, M.; Reif, W.; Schellhorn, G.; Stenzel, K.: VSE: Controlling the Complexity in Formal Software Development. In: *Proceedings Current Trends in Applied Formal Methods*, FM-Trends 98, LNCS 1641. SpringerVerlag, Boppard, Germany, 1999.

[HRU76] Harrison, Michael A.; Ruzzo, Walter L.; Ullman, Jeffrey D.: Protection in Operating Systems. In: *Commun. ACM*, volume 19(8):pp. 461–471, 1976.

[HV00] Hagimont, D.; Vandewalle, J.J.: JCCap: Capability-based Access Control for Java Card. In: Watson, A.; Domingo-Ferrer, J.; Chan, D., editors, *Proceedings of CARDIS - Smart Card Research and advanced Applications*. 2000.

[ITS91] Information Technology Security Evaluation Criteria (ITSEC). 1991. Commission of the European Communities.

[Jür01] Jürjens, Jan: Secrecy-preserving Refinement. In: *Formal Methods Europe (International Symposium)*, volume 2021 of *LNCS*, pp. 135–152. Springer-Verlag, 2001.

[KAT00a] Karger, A.; Austel, V.; Toll, D.: A new mandatory security policy combining secrecy and integrity. Technical Report RC 21717, IBM Research Division, T.J.Watson Research Center, Yorktown Heights, NY, 2000.

[KAT00b] Karger, A.; Austel, V.; Toll, D.: Using a mandatory secrecy and integrity policy on smart cards and mobile devices. In: *(EUROSMART) Security Conference*, pp. 134–148. Marseille, France, 2000. RC 21736 available at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

[KUW95] Koob, F.; Ullmann, M.; Wittmann, S.: The Formal VSE Development Method - A Way to Engineer High-Assurance Software Systems. In: *Proceedings of the 11th Computer Security Application Conference*, pp. 196–204. IEEE Computer Society Press, New Orleans, 1995.

[Lam94] Lamport, L.: The temporal logic of actions. In: *ACM Transactions on Programming Languages and Systems*, 16(3). 1994.

[LARS07] Langenstein, B.; A., Nonnengart; Rock, G.; Stephan, W.: Verification of Distributed Applications. In: *Computer Safety, Reliability, and Security, 26th International Conference, SAFECOMP 2007, Nuremberg, Germany*, LNCS. Springer, 2007.

[Las02] Lassmann, G.: Some results on robustness, security and usability of biometric systems. In: *IEEE International Conference on Multimedia and Expo*. Lausanne, 2002.

[Las06] Lassmann, G.: Kriterienkatalog zur Vergleichbarkeit biometrischer Verfahren, Version 3.0. Technical report, TeleTrusT Deutschland e.V., AG6: Biometrische Identifikationsverfahren, Chamissostraße 11, 99096 Erfurt, Germany, 2006. TeleTrusT Deutschland e.V., AG6 Biometrische Identifikationsverfahren.

[LS06] Laßmann, Gunter; Schwan, Matthias: Vertrauenswürdige Chipkartenbasierte Biometrische Authentifikation. In: Dittmann, Jana, editor, *Sicherheit 2006: Sicherheit - Schutz und Zuverlässigkeit, Beiträge der 3. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V.*, volume 77 of *LNI*, pp. 66–77. GI, 2006.

[McC87] McCullough, D.: Specifications for multi-level security and hook-up property. In: *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 161–166. 1987.

[McL94a] McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: *Proc. IEEE Symposium on Research in Security and Privacy*, pp. 79–93. 1994.

[McL94b] McLean, J.: Security models. In: Marciniak, J., editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

[Mea96] Meadows, Catherine: The NRL Protocol Analyzer: An Overview. In: *Journal of Logic Programming*, volume 26(2):pp. 113–131, 1996.

[Mer98] Merz, Stephan: A user's guide to TLA. In: Cassez, F.; Jard, C.; Roux, O.; Rozoy, B., editors, *Modélisation et vérification des processus parallèles: Actes de l'école d'été*, pp. 29–44. Ecole centrale de Nantes, Nantes, France, July 1998.

[MMJP03] Maltoni, D.; Maio, D.; Jain, A.; Prabhaker, S.: *Handbook of Fingerprint Recognition*. Springer-Verlag New York, 2003.

[MMYH02] Matsumoto, T.; Matsumoto, H.; Yamada, K.; Hoshino, S.: Impact of artificial gummy fingers on fingerprint systems. In: *Proceedings of the SPIE*, Optical Security and Counterfeit Deterrence Techniques IV, pp. 275–289. 2002.

[MvOV97] Menezes, A.J.; van Oorschot, P.C.; Vanstone, S.A.: *Handbook of Applied Cryptography*. CRC Press Series on Discrete Mathematics and Its Application, 1997.

[NPW02] Nipkow, T.; Paulson, L.C.; Wenzel, M.: *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[NTN02] Nanavati, S.; Thieme, M.; Nanavati, R.: *Biometrics - Identity Verification in a Networked World*. John Wiley & Sons New York, 2002.

[ORR+96] Owre, S.; Rajan, S.; Rushby, J.M.; Shankar, N.; Srivas, M.: PVS: Combining specification, proof checking, and model checking. In: *Alur, R. and T.A. Henzinger (editors): Computer Aided Verification*, volume 1102 of Lecture Notes in Computer Science, pp. 411–414. 1996.

[OWL03] Oheimb, David von; Walter, Georg; Lotz, Volkmar: A Formal Security Model of the Infineon SLE 88 Smart Card Memory Management. In: *Proc. of the 8^{th} European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *LNCS*. Springer, 2003.

[Pau98] Paulson, Lawrence C.: The Inductive Approach to Verifying Cryptographic Protocols. In: *Journal of Computer Security*, volume 6:pp. 85–128, 1998.

[PP01] Protection Profile - Secure Signature-Creation Device. 2001. European Committee for Standardization.

[RE02] Rankl, W.; Effing, W.: *Handbuch der Chipkarten: Aufbau - Funktionsweise - Einsatz von Smart Cards*. Hansa Verlag München Wien, 4th edition, 2002.

[Roc04] Rock, G: *Formal Methods for Real-Time Requirements Engineering*. Ph.D. thesis, Universität des Saarlandes, Germany, 2004.

[Roy87] Royce, W. W.: Managing the development of large software systems: concepts and techniques. In: *Proceedings of the 9th international conference on Software Engineering ICSE '87*, Lecture Notes in Computer Science. IEEE Computer Society Press, March 1987.

[RSG+00] Ryan, P.; Schneider, M.; Goldsmith, M.; Lowe, G.; Roscoe, B.: *Modeling and Analysis of Security Protocols*. Addison Wesley, 2000.

[RSW+99] Rock, Georg; Stephan, Werner; Wolpers, Andreas; (Hrsg.), R. Berghammer; (Hrsg.), Y. Lakhnech: Modular Reasoning about Structured TLA Specifications. In: *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pp. 217–229. Springer, 0 1999.

[Rus92] Rushby, John: Noninterference, transitivity, and channel-control security policies. Technical report, SRI International Computer Science Laboratory, dec 1992.

[San06] Santen, Thomas: Stepwise Development of Secure Systems. In: Janusz Górski, editor, *Computer Safety, Reliability, and Security, 25th International Conference, SAFECOMP 2006*, volume 4166 of *LNCS*, pp. 142–155. Springer, 2006.

[SBB+06] Sprenger, Christoph; Backes, Michael; Basin, David; Pfitzmann, Birgit; Waidner, Michael: Cryptographically sound theorem proving. Cryptology ePrint Archive, Report 2006/047, 2006. http://eprint.iacr.org/.

[Sch96] Schneier, B.: *Applied Cryptography – Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2nd edition, 1996.

[Sch07] Schwan, Matthias: An extended model of security policy for multiapplicative smart cards. In: *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pp. 226–233. ACM Press, New York, NY, USA, 2007.

[Ser89] Catalogue de Critères Destinés à évaluer le Degré de Confiance des Systèmes d'Information. Service Central de la Sécurité des Systèmes d'Information, 1989.

[Sny81] Snyder, Lawrence: Formal Models of Capability-Based Protection Systems. In: *IEEE Trans. Computers*, volume 30(3):pp. 172–181, 1981.

[SRS+00] Schellhorn, G.; Reif, W.; Schairer, A.; Karger, P.; Austel, V.; Toll, D.: Verification of a Formal Security Model for Multiapplicative Smart Cards. In: *Proceedings of the 6th European Symposium on Research in Computer Security*, LNCS 1895, pp. 17–36. Springer, 2000.

[SRS+02] Schellhorn, G.; Reif, W.; Schairer, A.; Karger, P.; Austel, V.; Toll, D.: Verified Formal Security Models for Multiapplicative Smart Cards. In: *Journal of Computer Security*, volume 10(4):pp. 339–368, 2002.

[SRSS99] Schellhorn, G.; Reif, W.; Schairer, A.; Stephan, W.: A Generic Security Model for Multiapplicative Smart Cards - Final Report of the SMaCOS project. Technical report, Federal Office for Information Security, Bonn, 1999.

[SS99] Schneier, B.; Shostack, A.: Breaking Up Is Hard to Do: Modeling Security Threats for Smart Cards. In: *USENIX Workshop on Smart Card Technology*, pp. 175–185. Usenix Press, 1999.

[SSGS00] Scheuermann, D.; Schwiderski-Grosche, S.; Struif, B: Usability of biometrics in relation to electronic signatures. Technical Report Nr. 118, GMD, 2000.

[Ste91] Sterne, Daniel F.: On the Buzzword "Security Policy". In: *IEEE Symposium on Security and Privacy*, pp. 219–231. 1991.

[Sut86] Sutherland, D.: A Model of Information. In: *Proceedings of the 9. NIST National Computer Security Conference, National Bureau of Standards, National Computer Security Center*, pp. 175–183. 1986.

[TKZ02] Thalheim, L.; Krissler, J.; Ziegler, P.M.: Body Check: Biometric Access Protection Devices and Their Programs put to the Test. In: *c t - Magazin für Computertechnik*, volume 10:p. 114, 2002.

[WSE04] Waldmann, U.; Scheuermann, D.; Eckert, C.: Protected transmission of biometric user authentication data for oncard-matching. In: *Proceedings of the 2004 ACM Symposium on Applied Computing*, pp. 425–430. ACM Press New York, 2004.

[Zha00] Zhang, D.: *Automated Biometrics: Technologies and Systems.* Kluwer Academic Publishers, 2000.

Index

Access Categories, 37
Access Classes, 37
Access Control Models, 18, 67
    Discretionary Policies, 18, 25
    Mandatory Policies, 18, 25
    Role-Based Policies, 18
Access Level, 37
Advanced Electronic Signature, 10
AeSig, 10
AirPts, 9
Attacker Model (Dolev/Yao), 71

BasicCard, 2, 33
Biba Security Model, 37, 68
Biometric
    Enrollment, 11
    Identification, 11
    System, 11
    Template, 11
    Verification, 11
BioSig, 10, 12
BLP Security Model, 37, 67

Card
    Application Provider, 20
    Hardware Manufacturer, 20
    Holder, 19
    Interface Provider, 20
    Issuer, 20
    Software Manufacturer, 20
Channel Programs, 41
CommC, 101
Common Criteria (CC), 4, 10, 76
Confinement Property, 38

CTPEC, 75

eHealth, 14
Enrollment, 11
Evaluation Assurance Level (EAL), 77
ExtAppl, 102

False Acceptance Rate (FAR), 12
False Rejection Rate (FRR), 12
FIFO Principle, 97, 111
FIPS, 75
Formal Analysis of
    Security Policies, 67
    Security Protocols, 70
Formal Methods, 66

Inductive Model (Paulson), 146
Information Flow Models, 18, 67
ISO 7816 smart cards, 28
ITSEC, 4, 76

Java
    Applets, 28
    Card, 28
    Language security, 29
    Sandbox Model, 30
JavaCard, 2

KeyGen, 100

Liveness Properties, 94

Mode of Operation
    issuer oriented, 24, 34
    issuer oriented with third party, 24, 34

user oriented, 24, 33
MultAC, 7
Multi Level Security (MLS), 19
Multilevel access control, 7
Multos, 2, 30
Mutual Authentication Protocol, 128, 147
Authenticity theorem, 132
Confidentiality theorem, 134
Regularity lemma, 132
Unicity theorem, 134

NIST, 75

Observer
Mapping, 151
Models, 150

Protection Profile (PP), 12, 76

QeSig, 10
Qualified Electronic Signature, 10

Safety Properties, 70, 73, 94
SCD, 11
Security, 17
Environment, 64
Evaluation Criteria, 75
Functions (SF), 17
Model, 18
Objectives, 23
Objectives (SO), 18
Policy (SP), 18, 63
Completeness, 65
Consistency, 65
Effectiveness, 65
Multi-applicative smart cards, 23
Policy Model, 18
Properties, 71, 73
System, 17
Target (ST), 76
Threats (ST), 17, 19
Triangle, 64
Simple Security Property, 38
SMaCOS, 31
Social Engineering, 64
SSCD, 10
SSCM, 96, 148
Confidentiality Property CO, 102
Extended (eSSCM), 154
Integrity Property IN, 103
Non-duplicating Property ND, 102
Protocol Property (eSSCM), 161
Quality Property QK, 102
Strength Of Function (SOF), 10

Tamper-resistance, 64
Target of Evaluation (TOE), 76
TCSEC, 75
Threat Classification Tree, 21
Threat Model, 21
TLA, 87, 148
Action Formulas, 89
Auxiliary Variables, 96
Fairness Conditions, 92
History Variables, 96
Liveness Properties, 94
Safety Properties, 94
State Formulas, 88
Temporal Formulas, 90
TOE Security Functions (TSF), 77

VSE-II, 104
Development Graph, 105
Heuristics, 142
Security Model, 105
VSE-SL, 104

Windows for Smart Cards, 2

Erklärung

Ich erkläre hiermit, dass

- ich die vorliegende Dissertationsschrift „Specification and Verification of Security Policies for Smart Cards“ selbstständig und ohne unerlaubte Hilfe angefertigt habe;
- ich mich nicht anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze;
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin Nr. 34 vom 17.01.2005 (zuletzt geändert am 13.02.2006) bekannt ist.

Berlin, den 15. Oktober 2007 Matthias Schwan